(1

Architecture and Design for a Laser Programmable Double Precision Floating Point Application Specific Processor

THESIS

John Henry Comtois
Captain, USAF

AFIT/GE/ENG/88-5

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GE/ENG/88-5

Architecture and Design for a Laser Programmable Double Precision Floating Point Application Specific Processor

THESIS

John Henry Comtois
Captain, USAF

AFIT/GE/ENG/88-5

AFIT/GE/ENG/88-5

Architecture and Design for a Laser Programmable Double Precision Floating Point Application Specific Processor

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

John Henry Comtois, B.S.E.E.

Captain, USAF

December 1988

I wish to thank my wife, Susan, for putting up with the year- and-a-half separation while I attended AFIT.

I would also like to thank my thesis advisor, Capt. Richard Linderman for his assistance and insistence during the course of this research. I also extend my thanks to the summer quarter 1988 "Introduction to Computer Architecture" class for their comments and suggestions on the project. I would like to thank the other members of the VLSI group: Captain Keith Jones, CPT Michael Dukes, CPT Erik Fretheim, CPT James Kainec, and especially Captain John Tillie, for their assistance and advice. I would also like to thank the members of my thesis committee, Lt. Col. Bisbee and Major DeGroat.

Table of Contents

Page

List of Figures

Figure Page

List of Tables

AFIT/GE/ENG/88-5

Abstract

Numerous signal processing systems in the Department of Defense and industry would benefit from a microprocessor tailored to their specific applications. This thesis effort describes the architecture and design for a 64 bit application specific processor (ASP) which combines the power of double precision floating point hardware with the flexibility of a laser programmable microcode store. This floating point ASP (FPASP) contains a variety of circuits which efficiently perform digital signal processing and other applications requiring double precision floating point arithmetic.

This thesis describes a new rapid rapid prototyping methodology for ASPs. The user provides an algorithm which is translated into microcode, tested on a software model, and then cut into the laser PROM of a blank FPASP chip. With this methodology the prototyping of an ASP can be completed in a matter of days, and no hardware design is involved. The programmed FPASP can then be mounted on a circuit board and placed in a host processor to act as a hardware accelerator for computationally intense programs. The FPASP also supports a macro assembly language which can be partially user-defined. So the FPASP can be tailored to higher level applications such as operating system support. Keywords; Computer architecture, Very Large Scale Integration, [illegible] [illegible] [illegible]. (KR)

The FPASP has been designed to support common software structures. It contains registers for loop indexing, and addressing into matrices. The FPASP also contains a subroutine stack 16 words deep, which can be extended into the external memory for an additional 1023 words to support recursive microcode routines. The FPASP will contain 180,000 CMOS transistors on a chip 0.35 inches on a side. It is designed to operate at 25 MHz, and at that speed it will be capable of performing 25 million floating point operations a second.

The FPASP architecture consists of two 32 bit processors which can operate independently for integer operations, or in tandem for double precision floating point operations. Overlapping register sets on each datapath and ties between the two datapaths provide a high degree of interconnectivity, allowing efficient internal data transfer between the 66 32 bit registers and the processing circuits.

Architecture and Design for a Laser Programmable Double Precision Floating Point Application Specific Processor

I. *Introduction*

1.1 **Background**

Continuing advances in integrated circuit fabrication make possible ever smaller feature sizes, allowing more circuitry to be placed on a chip. The hardware required to perform double precision floating point arithmetic, which is often packaged as a separate "coprocessor," can now be placed on the same chip as the basic processing hardware, and the chip can be kept down to a size that allows reasonable yields.

Recent research in laser technology at the Massachusetts Institute of Technology Lincoln Laboratory has shown that circuits can be designed and fabricated using standard processes and then altered after packaging. A laser programmable read-only memory (LPROM) can contain the microcode program used in an application specific processor (ASP). The design of an LPROM based on this technique is the topic of a concurrent thesis by Capt. Tillie [Til88]. The hardware needed to program the LPROM and the software needed to automate the process have also been developed as part of that effort.

Research projects at AFIT have produced a single precision floating point multiplier and a library of sub-circuits for a bit-slice application specific processor [Gal87]. These circuits were part of a methodology for producing prototypes of application specific processors by assembling cells from a cell library, then fabricating the circuit. Their purpose was to decrease the time required for designing these types of circuits. A test processor based on this methodology was designed and fabricated in 3 micron CMOS. The single precision floating point multiplier has also been fabricated.

AFIT is currently researching digital signal processing for space surveillance applications. One of the algorithms used for digital signal processing is Kalman filtering. This algorithm is computationally intensive. It would be more efficient to have a dedicated piece of hardware to carry out the computations rather than use a general purpose computer. This makes the algorithm a good choice for implementation with an ASP. To obtain the most accurate results, a double precision data format has been chosen. Hardware for double precision floating point arithmetic is currently being designed.

These projects set the stage for this thesis, which will bring them together with the technological advances mentioned in the design of a laser programmable processor to meet the needs of the Kalman filter and similar computational problems.

1.2 Problem Statement

This research will create a processor architecture and VLSI design which will support double precision floating point arithmetic and will be micro-programmable after fabrication for specific applications.

1.3 Objectives

The objective of this thesis is to take advantage of the new 1.2 micron CMOS technology to show that a double precision floating point ASP can be put on a single chip and contain enough proc ssing hardware to be useful for a wide range of applications. In addition, laser programming technology will be employed to allow the ASP to be tailored to the specific application *after* fabrication. An architecture will be developed for a laser programmable, double precision floating point application specific processor (FPASP). The FPASP will form the basis of a new rapid prototyping methodology wherein the "fabricate a new chip for each application" paradigm previously applied to ASPs will be replaced with "fabricate a generic ASP in quantity then program copies of it as needed".

The FPASP will support the software structures used in matrix algebra microcode. These structures include nested loops, recursion, and data stored in row-major order. Microcode written for the FPASP will demonstrate the usefulness of the architecture for handling a variety of applications which use these structures. The FPASP will support an assembly language which can be partially user-defined.

The FPASP must also be able to perform a Kalman filter routine without laser programming, in case the laser hardware is not ready when the first application must be done.

1.4 Scope

1.4.1 **Scope of the Thesis Report.** This report will detail the architecture and design of the FPASP, exclusive of the floating point hardware and the laser programmable memory

[May88,Fre88]. These circuits are the subject of other projects concurrent with this thesis effort, and so will only be briefly discussed in this report.

Microcode written as part of this effort and as a class project for the EE588 class from the summer quarter 1988 will be used to demonstrate how the hardware supports software structures. In addition to this microcode, the microcode and hardware which supports an assembly language will also be discussed.

Models of the FPASP used for verifying the design of the subcells will be discussed. A high-level structural model written in the VHSIC Hardware Description Language (VHDL) will be listed. The VHDL model will become part of the new methodology for the rapid prototyping of ASPs. The methods used to verify the proper operation of the subcircuits will also be covered.

1.4.2 **Scope of the FPASP Architecture.** The scope of problems for which the FPASP was designed is the general class of problems which manipulate matrix type databases stored in row-major order. A library of code to be stored permanently on the FPASP will contain subroutines which perform basic matrix algebra operations such as dot product and matrix-matrix multiply.

The FPASP most efficiently supports two data representations: 32-bit integers and 64-bit IEEE double precision floating point numbers. The built-in floating point hardware will include a multiplier and an adder/subtractor. The FPASP is designed to directly support nested loops to a depth of six, and recursive subroutine calls to a depth of 1040.

Three features extend the scope of the FPASP concept beyond the original ASP concept. One is the ability to write microcode into the FPASP's laser programmable ROM (LPROM) after it is fabricated, rather than fabricating an entirely new chip each time an ASP is needed. Another is an assembly language which is partially coded into the FPASP's fixed ROM library and partly user-definable via the LPROM and a laser programmable opcode map. The third is a subroutine stack the extends into the external memory. These features increase the scope of problems the FPASP can handle.

1.5 Summary of Current Knowledge

Research on the design of integer ASPs supplied some of the macrocells used in the FPASP. The detailed operation of these cells has been determined by reading Capt. Gallagher's thesis, looking at the cells' magic descriptions, and by reverse engineering their logical structure from SIM files. The SIM extractor was written by CPT Dukes and tested on some of the cells from the ASP library [Duk88]. An integer ASP test chip which uses these cells has been fabricated by MOSIS and is available for testing.

Microcode routines were written by Capt. Linderman for a single precision floating point ASP based on the Capt. Gallagher's integer ASP architecture. These routines were available for conversion to FPASP microcode. In addition to these, routines written as EE588 class projects were available to point out the strengths and weaknesses of the FPASP as its design evolved from Capt. Linderman's early design.

A large number of cells also exist for the Winograd Fourier Transform (WFT) processor chips [She86]. These chips not only provided pad cells for the FPASP, but also provided design and layout experience during a class project which involved creating the WFT17 chip from the similar but smaller WFT16 chip.

Another chip laid out for a previous thesis effort was a laser programmable comparator [Spa86]. That chip was reconfigured to be used on a proposed ASP board for the Sun Microsystems workstations [Jav87]. That chip has been fabricated as a MOSIS TinyChip [MOS88]. Work on that chip provided experience as a background to this thesis effort.

1.6 Assumptions

It is assumed that the floating point hardware projects will be completed in time to integrate those cells into the FPASP as a part of this thesis effort. Floorplanning of the FPASP indicated how much space is available for those cells. These estimates were based on the completed core adder array of the floating point multiplier. The interface for the floating point hardware is fixed, so if they are not done in time for integration as part of this effort, it will take only a few days' work to complete their integration with the rest of the FPASP.

Another assumption is that the equipment for the laser programming will be in place and operating when it comes time to program the FPASP. It was assumed from the start that laser programming would work, based on reports from Lincoln Laboratories. It is also assumed that the programming speed of the laser hardware will be increased before any actual programming of FPASPs in quantity will take place. Presently, speeds proposed by Capt. Tillie indicate a programming time for 256 64-bit words of laser PROM of 12 hours.

It is also assumed that the memory chips and printed circuit board to be used with the FPASP will be capable of supporting a single cycle read or write.

1.7 Approach

1.7.1 Approach to Architectural Specification.

Figure 1.1 shows a hierarchy of design which will be referred to throughout this paper. The size of the text indicates the areas of emphasis for this thesis. The formulation of the FPASP architecture followed neither a top-down approach from software, nor a bottom up approach from hardware. The top-down approach cannot take into account the problems encountered in the design of hardware, which could result in an architecture that cannot be realized in silicon. On the other hand, the bottom up approach from hardware cannot efficiently support a software routine which has not been written yet, which means the programmer would have to use whatever the designer decided was good enough or what would fit on the chip.

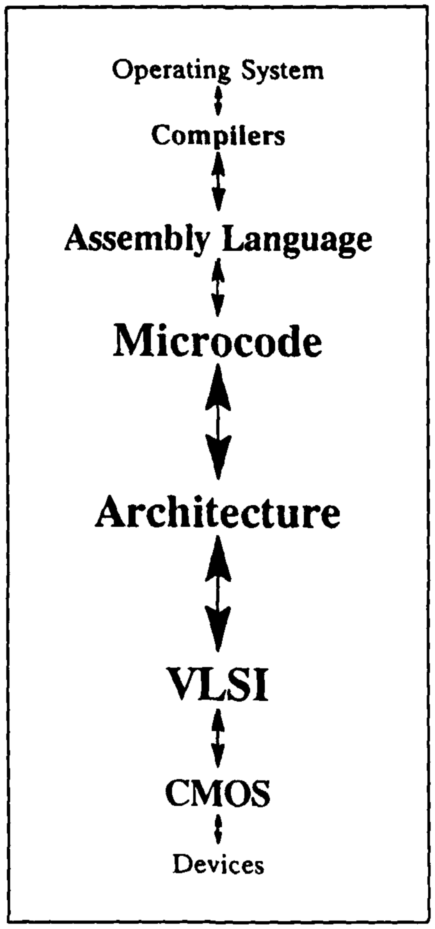


Figure 1.1. Theme Figure.

The approach chosen for the FPASP was an iterative one: microcode was written at various stages of architectural development. This provided feedback, pointing out hardware that was needed but not available, or hardware that was available but was superfluous or not used often enough to justify the area it occupied. At the same time, floorplanning of the FPASP was carried out. This provided size estimates of the hardware needed to support the microcode. Thus the development of the FPASP architecture proceeded with feedback coming from both sides which guided it towards an efficient compromise between what would be nice to have for the software, and what could be fit onto a chip with reasonable yields.

Throughout the evolution of the architecture and development of the macrocells, design for testability was included. The approach to design for testability followed closely the approach taken for the control section of the WFT17 chip. Emphasis was placed on making the microcode sequencer control section more observable and controllable.

1.7.2 **Approach to Chip Design.** Before actual cell design was attempted, a floorplan was generated to get an idea of size and placement of the cells. The sizes of cells were based on existing ASP and LPROM cells, and on the completed portion of the floating point multiplier.

After floorplanning, a search was made of the existing cell libraries to see which cells could be reused directly or with slight modification. Whenever possible, cells which had to be designed for the FPASP were based on existing cells or on a set of standard parts gleaned from those cells.

Early on it was obvious that the FPASP would need more control bits than could be efficiently held in microcode memory; so emphasis was placed on choosing microcode fields and encoding schemes which could be decoded with a minimum of extra circuitry. This affected the design of the cells and the ordering of the encoded choices in each microcode field.

1.8 Materials and Equipment

This thesis effort made use of the Berkeley Unix CAD tool set [Cal86]. This included the Magic layout tool, the Mextra transistor netlist extraction tool, and the Esim switch level simulator [Ter86]. Several CAD tools written at AFIT were also used. These included the CSTAT static design checker [Lin85], the GMAT microcode assembler [Hau87], and the circuit extractor mentioned above. Spice 3 was used for analog simulation of transistors based on models received from the fabricator on previous 1.2 micron CMOS fabrication runs [Qua86]. IEEE 1076 standard VHDL was used for the model [Fle88].

Hardware tools used included the ELXSI superminicomputers (BSD and ICC), the Vax 11/785 computers (SSC and CSC), a Sun 3/160 workstation (Mercury), an AED767 graphics display with a Summagraphics digitizing pad, and laser printers. All of the machines listed above were available at AFIT. I also used my home system, a Heathkit H100/IBM PC compatible with a 1200 baud modem.

The MOS Implementation Service (MOSIS) will be used for the fabrication of the FPASP, with funding from the Defense Advanced Research Projects Agency (DARPA).

1.9 Order of Presentation

The following chapters cover different aspects of the research. For each chapter a version of Figure 1.1 is used to show which areas will be concentrated on. In these figures the inner brace shows the main area, and the outer brace shows areas which are also affected or which generated feedback.

Chapter 2 presents an analysis of the problems and requirements that affected the FPASP architecture and VLSI design. Chapters 3 through 5 present the research and how it was carried out. Chapter 3 covers the details of the FPASP architecture and how feedback from hardware and software research affected it.

Chapter 4 examines the hardware designed to implement the FPASP architecture. Block diagrams are used to show how the various macrocells work together to embody the architecture in a real chip layout. Microcode routines are presented to show how these macrocells are designed to support software structures.

Chapter 5 details the low-level VLSI design and modeling of the FPASP. This includes verification of the design at both the transistor and macro-cell level.

Chapter 6 discusses the results of the FPASP project. The efficiency and variety of the microcode written for the FPASP will demonstrate its usefulness. The projected performance of the FPASP will be compared with the performance of some other machines to give an idea of what the computing power of the FPASP will be.

Finally, Chapter 7 presents some conclusions about the overall FPASP thesis effort, and lists some suggestions for future projects related to the FPASP.

II. Problem Analysis

2.1 Introduction

This chapter discusses the design criteria that the FPASP architecture had to support, and the technological constraints which affected the design of the FPASP chip. The solutions to these problems are the subject of Chapters 3 through 5.

2.2 Efficient Dot Product Routine

The first application for the FPASP chip will be to perform digital signal processing in the form of Kalman filtering, on a VMEbus-compatible board in a Sun workstation. Therefore, the FPASP must efficiently support matrix algebra, which is used extensively in the Kalman filtering algorithm.

A basic operation in matrix algebra is the dot product of two vectors. An example of a dot product is shown below.

$[246] \cdot [135]^T = 2 * 1 + 4 * 3 + 6 * 5 = 44$

The elements of each vector are multiplied together and accumulated into a final result which is a scalar. This routine is at the core of a larger routine for multiplying two matrices together, as seen in the flowchart in Figure 2.1.

The matrix multiply routine is a double-nested loop, with a dot product in the inner loop. This means that for square matrices with n vectors, the dot product routine must be done n^2 times, with each dot product performing n multiplies and n-*1* accumulates; thus n^3 multiplies and $n^2[n-1]$ accumulates must be done for each matrix-matrix multiplication. The matrix multiply is a basic matrix algebra operation; therefore, the speed of the dot product routine will determine the overall speed, within a constant factor, of any algorithm which depends on it.

An efficient dot product routine was one of the basic problems to be solved by the FPASP architecture. It had to provide the datapaths and functionality to perform these two operations, multiply and accumulate, on double precision floating point numbers, including moving the data on and off the chip, in as few clock cycles as possible.

A processor architecture with single precision floating point hardware was designed by Capt. Linderman for a class project in microcode design. A register-level description of this processor appears in Figure 2.2. The winter quarter 1988 EE588 class project was to write Kalman Filter microcode for this processor.

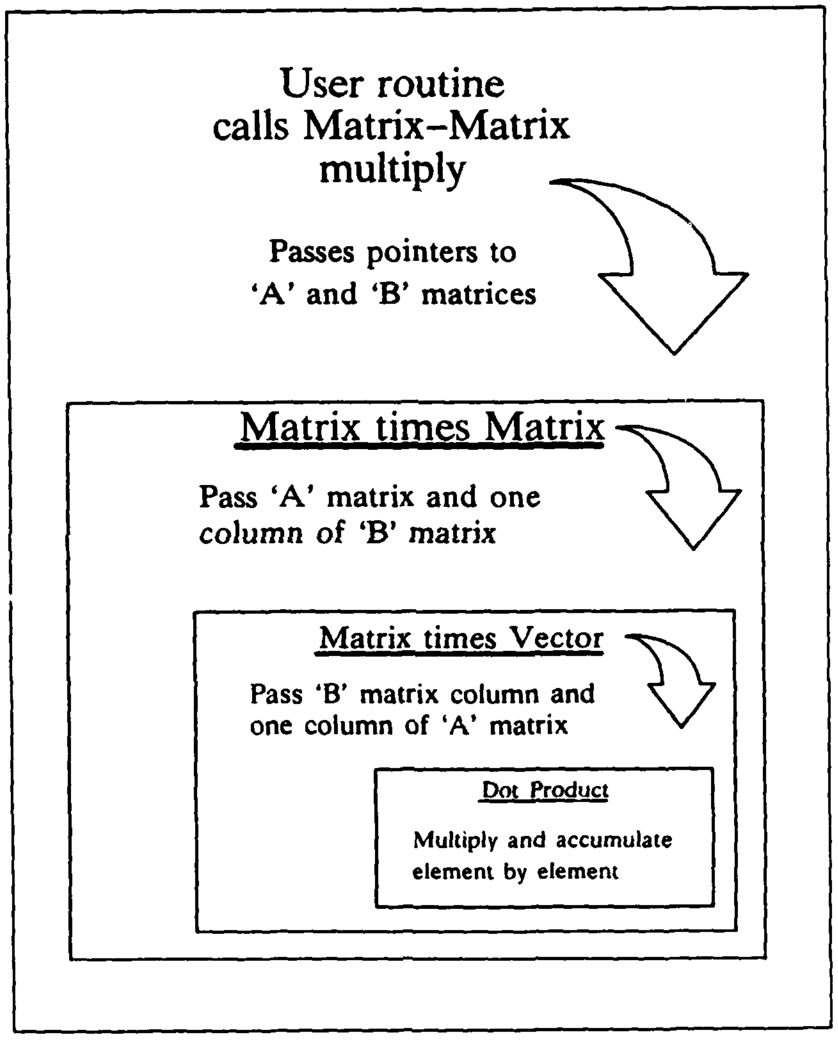

User routine calls Matrix–Matrix multiply
Passes pointers to 'A' and 'B' matrices
Matrix times Matrix
Pass 'A' matrix and one column of 'B' matrix
Matrix times Vector
Pass 'B' matrix column and one column of 'A' matrix
Dot Product
Multiply and accumulate element by element

Figure 2.1. Matrix-Matrix Multiply Flow.

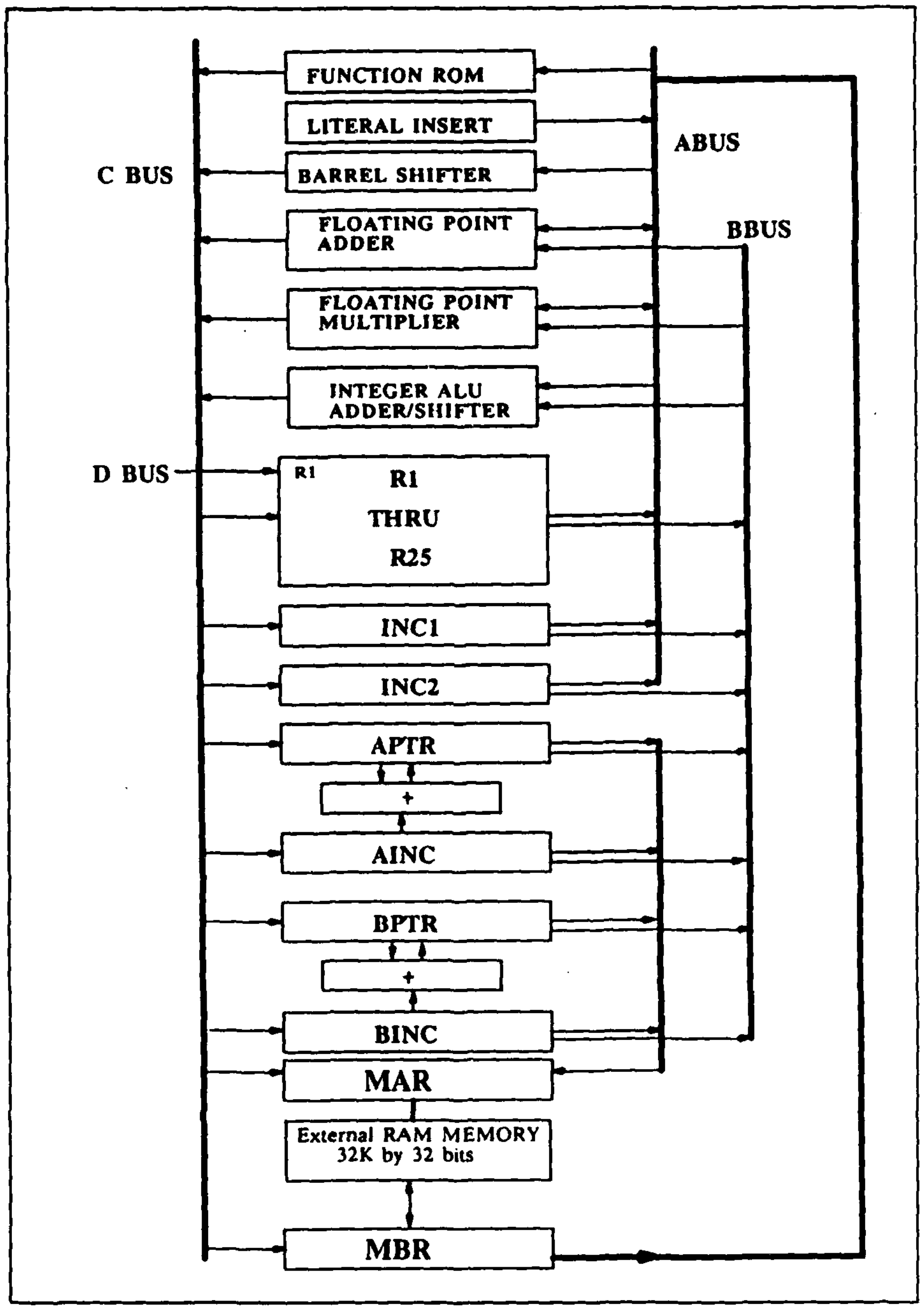


Figure 2.2. SPASP Register-level Description.

One important feature of this architecture was specifying that the floating point hardware would have two clock cycles to compute. The microcode must contend with this pipelined data flow, but the processor gains efficiency by performing more operations in each clock cycle. The length of the clock cycle can be decreased by giving the floating point processors two clock cycles instead of one. This allows two integer operations to be performed since all of the integer operations can be completed in the shorter cycle time.

So, while a floating point result is being computed, the rest of the processor can be doing other things. For example, the floating point adder can be loaded in one cycle, and the multiplier in the next; the third cycle can then be used to unload the adder and the fourth cycle to unload the multiplier. This back and forth loading and unloading is used by the dot product routine to keep both floating point processors in continuous operation; while at the same time computing the addresses for the operands.

2.2.1 **Pipelined Architecture.** Pipelined structure such as the two cycle computation time for the floating point hardware is common in processor design where one or more computations take much longer than basic processing functions such as calculating data addresses. In the case of the SPASP, the delay is worked out by the microcode, but this problem also occurs in non-microcode-driven architectures. In those cases, pipelining is used when there are more processing circuits than there are pins available to supply data to each one simultaneously.

An example of a processor pipelined for the second reason is shown in Figure 2.3 [Gun88]. The IQMAC chip performs single precision floating point arithmetic for digital signal processing. It uses a hardware pipeline to increase the number of operations being performed on each clock cycle. The chip uses 1.2 μm CMOS technology.

With that much hardware packed onto a chip, the number of separate processing circuits has outstripped the packaging technology. In this case, the package is a 256 lead flatpack, but there are five separate 32 bit arithmetic processors. The hardware pipeline allows the data to be passed back and forth through fewer pins: the processors are arranged one after the other, so only the first stage of the pipeline needs to be loaded, and only the last stage needs to drive out data.

Pipelining has also been used in the SPASP architecture, as seen in Figure 2.2. In the SPASP input data is placed on the A and B busses. Normally the result would be driven onto the C bus in the following clock cycle and loaded into a register. In the dot product routine, a multiply and addition must be done in each iteration of the loop. This is where a change in the architecture can decrease the length of the software 'pipeline.' If the multiplier results can be driven directly onto the A or B bus, the dot product accumulated so far can be driven onto the other bus and the two numbers loaded into the adder in the same cycle. This eliminates the time lost putting the multiplier result into a register from the C bus first and then driving it out on the A bus the next cycle.

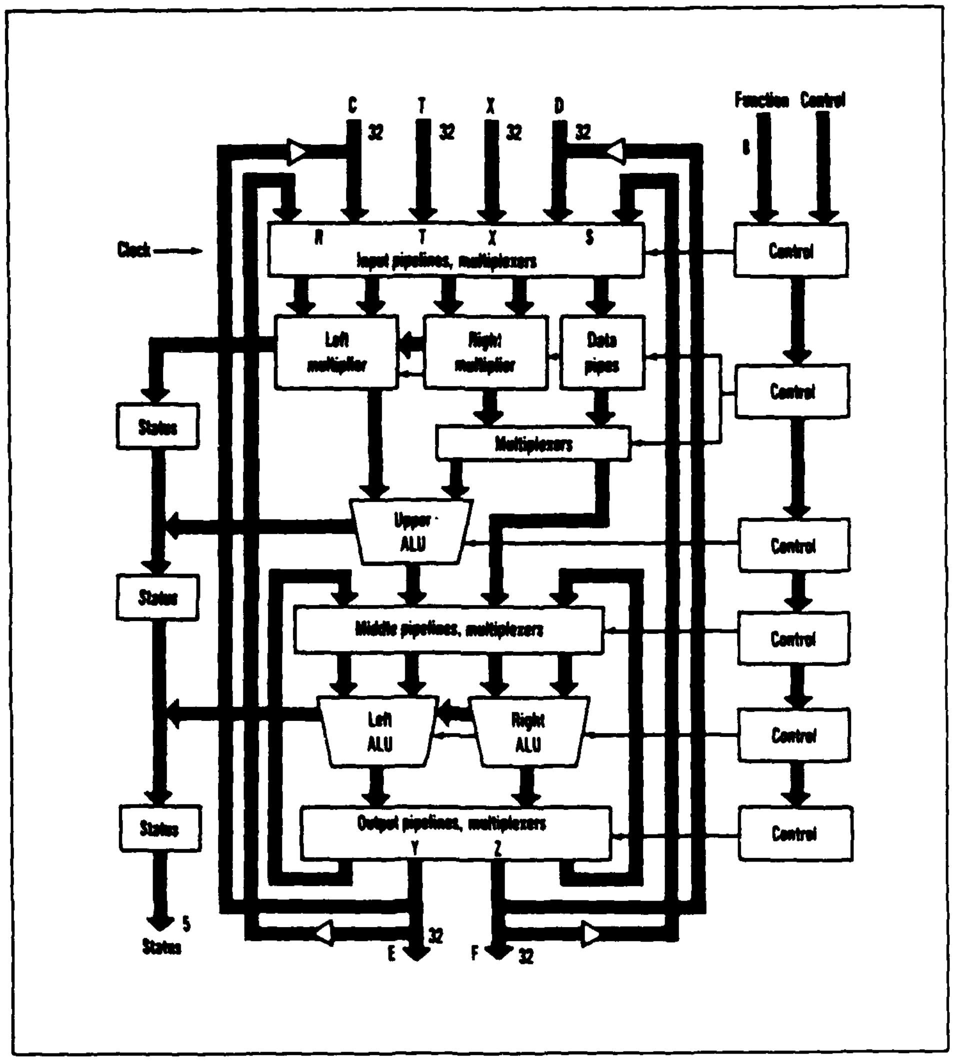


Figure 2.3. IQMAC Block Diagram. [Gun88]

2.3 Number Representations

The SPASP circuit proposed for Kalman filtering for the EE588 class was intended to be fabricated as an ASP based on the cell library created by Capt. Gallagher. That library only contained a single precision floating point multiplier.

A class project by Capt. Fretheim has indicated that a double precision multiplier can be made as a macrocell for an ASP using the 1.2 micron CMOS technology. This will allow the more precise multiplier to fit on a chip with enough room left over for the processing circuitry. The IEEE double precision floating point number representation is shown in Figure 2.4. The more precise data representation requires the ASP to support a data width of 64 bits, which is larger than the cells in the library were sized to support.

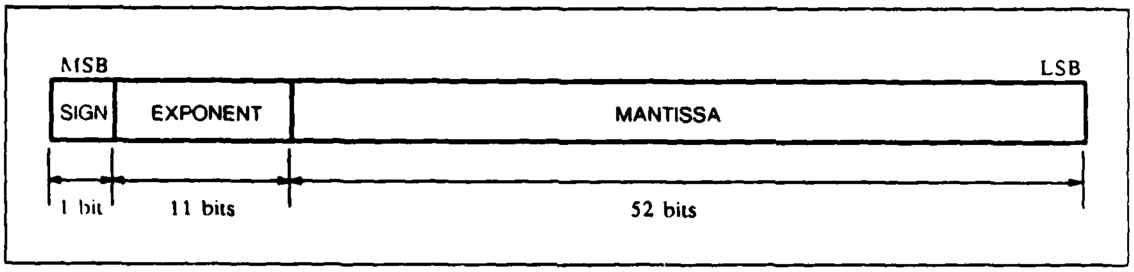


Figure 2.4. IEEE Double Precision Floating Point Format.

2.4 Support for Software Structures

The need to support the most efficient double precision dot product routine was one of the main problems to be solved by the FPASP architecture, but it was not the only one. Since the FPASP is intended to also support many as yet unknown algorithms, it must contain enough features to allow it to perform a wide range of functions. Two of these projected requirements are support for the software structures of nested loops and recursive routines. These additional features must not detract from the efficiency of the Kalman filtering routine.

Matrix data is commonly stored in row-major order: successive memory locations hold the elements of the array starting with the first element of the first row and continuing row by row to the last element of the last row. In matrix algebra routines, the data is often accessed column by column as well as row by row. In order to efficiently support this addressing mode, the register which holds the pointer to the data must be able to be incremented by the number of elements in a row. Since the FPASP must support generalized matrix algebra routines, this increment value must be easily varied.

In some microcode controlled processors such as the ASP, calls to a subroutine cause the return address to be saved onto a register stack. The depth of this stack determines how many calls can be made before the machine can no longer support that software structure. Some software programs require a routine to keep calling itself until a condition is met. Such recursive routines require a return address to be put on the stack for each such call, especially if the recursively called routine calls on other routines. A feature of recursive software code is that the number of calls is unknown to the calling routine. The cells designed for the ASP cell library limit the depth of the stack to the amount of space available on the chip for stack registers. For long recursion, chains it would be impractical to build a stack deep enough. The FPASP must be therefore be able to save the stack externally when it gets full.

2.5 Computer-Aided Design Restrictions

To design and lay out a chip as complex as the FPASP requires the use of software tools to verify the design and to draw the actual layout of the transistors. Such tools are part of the AFIT Computer Aided Design (CAD) environment. Some of the tools which proved most useful to the FPASP design process also placed some restrictions on the design. Two of the tools which had this effect were the GMAT and Esim tools. The GMAT tool converts microcode to binary machine code, and the Esim tool simulates a circuit at the switch level. The limitations of these two tools directly affected the architecture of the FPASP and the design of the FPASP circuitry.

The first limitation is that Esim cannot simulate circuits which contain certain feedback loops. Ongoing research projects at AFIT are solving this problem. The Fixrom and Nofeed programs get around the feedback problem by replacing the unsimulatable circuit description with one that contains no feedback loops. But this type of fix is limited to those circuit configurations which these programs are designed to recognize; so the basic problem with Esim remains and shows up when new circuit designs are simulated. This problem must be kept in mind when circuits are designed because a logically correct circuit that cannot be simulated is useless. Simulation of the completed FPASP circuit will be required before it is sent off for fabrication.

The second restriction is the maximum length of a microcode word that GMAT can handle, which is 128 bits. This was sufficient for the ASP prototype chip, which has a microcode word only 51 bits wide. The ASP was only a proof-of- concept prototype, so it had few control signals and the GMAT restriction had no effect. 51 bits allowed the ASP controller to be very horizontal:

almost every control line had a microcode bit assigned to it [Man82]. This eliminated most of the decoding that would then have to be done outside of the microsequencer.

GMAT does not support the number of control signals required for a large processor, so the horizontal design style cannot be used. This problem has two effects on the FPASP circuit: more decoding circuitry must be designed and laid out, and not all of the possible combinations of functions for each macrocell can be represented in the microcode fields.

2.6 CMOS Technology Limitations

Most of the limitations on circuit design are due to the CMOS technology, which is the only one supported by the AFIT CAD environment at this time. For example, the resistance of the channel formed under the gate of a MOS transistor limits the amount of current the device can pass. This gain is partly dependent on the fabrication process; but the designer decides on the sizes of the transistors and must therefore choose the sizes that best meet the circuit requirements. The sizing of transistors and the use of the Spice program to simulate those choices are covered in chapter 5.

Some limitations are due to the Very Large Scale Integrated circuit (VLSI) concept itself. This is the concept of putting as much circuitry as possible onto one chip to increase the speed and decrease the number of packages needed to perform a particular function. The number of devices on a chip has been doubling every two years, but the number of pins available per chip has not kept up that pace. The result is that the circuitry on the chip becomes less accessible from the outside.

This decrease in observability and controllability of VLSI circuits has increased the problem of testing them. This problem can be eased by adding extra circuitry to the design to increase the number of observable and controllable points in the circuit [Fuj85]. This design-for-testability (dft) problem must be addressed in the FPASP architecture and design.

III. Architecture

3.1 Introduction

The FPASP design process began with the architectural specification and a proposed microcode instruction set. This corresponds to the center of the diagram shown in Figure 3.1. The architecture evolved as feedback was generated from continuous research into both the software and hardware of the FPASP. This feedback allowed the final design to meet the needs of both sides, above and below the architecture level. Specific examples of how these feedbacks affected the architecture will be shown throughout this chapter. The result is a machine that solves the problems discussed in Chapter 2.

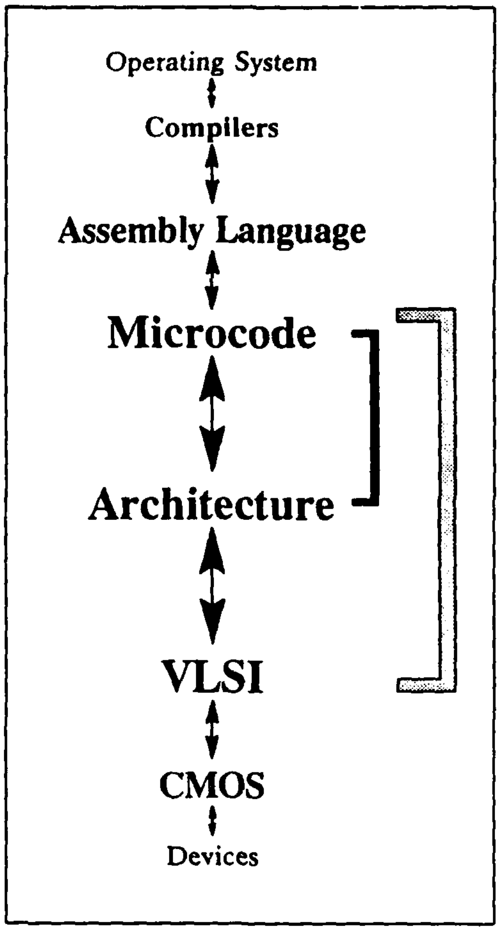


Figure 3.1. Design Areas Covered in this Chapter

3.2 Data Representations

Research for this thesis began after the decision was made to go with the double precision hardware for the Kalman filter ASP. The new ASP is called the Floating Point ASP (FPASP) to distinguish it from the integer ASP chip designed by Capt. Gallagher, and Capt. Linderman's single precision ASP (SPASP) architecture.

The first problem raised by going to double precision was how to represent 64 bit data inside the FPASP. Since the existing ASP cells are bit-slice, the easiest solution would be to have the data paths 64 bits wide. But this means using 64 bit data even when not doing floating point operations.

Most of the integer operations done in the microcode to support the floating point operations set up loops and calculate addresses. These operations do not require 64 bit words, and the extra bits just slow down the hardware. For example, the addresses used by the FPASP are 20 bits long, so there is no real need for a 64 bit word from the addressing point of view. Also, the first intended use for the FPASP is in a host processor with a 32 bit bus, so 32 bits is a good choice from an interfacing point of view.

A 64 bit integer representation is also difficult to support in hardware. For example, the ALUs in the FPASP use a carry-select adder. The ripple delay through this adder is the worst case delay in the ALU; extending the adder to 64 bits would slow down the entire machine's clock frequency. So using 64 bit integers would not be a good design choice. Also, with two 32 bit datapaths there is the possibility of doing two integer operations in the same clock cycle.

The 32 bit integer representation was chosen for the FPASP, along with 64 data paths on and off the chip. The mapping of the double precision format onto the two FPASP datapaths is shown in Figure 3.2.

3.3 General Architectural Features

Given the 32 bit internal data representation and 64 bit I/O width, there can be two separate datapaths in the FPASP architecture: upper and lower, both 32 bits wide. To support this architecture, the external memory was split into upper and lower halves, each with its own separate address lines from the FPASP. The basic architecture thus became two SPASPs which work together to manipulate floating point numbers, or separately to do software support such as looping and branching.

To get the two 32 bit datapaths, the SPASP register set and ALU were mirrored on each side. This simplified the physical layout and the design of the chip; an important factor given the

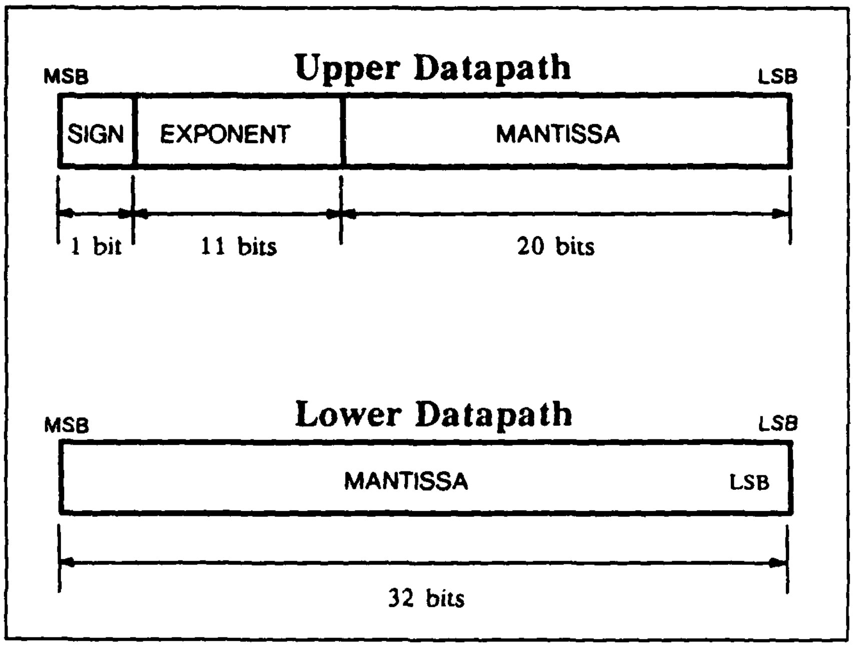


Figure 3.2. Double Precision Representation Inside the FPASP.

limited amount of time available for layout and design verification. The mirror image attribute of the FPASP can be seen in the register-level description in Figure 3.3, and in the floorplan of the chip. A simplified floorplan is shown in Figure 3.4.

Not all of the hardware was mirrored, however. Some of the hardware is only needed on one or the other datapath, or a piece of hardware may be so large that only one can be afforded in the FPASP. For example, the function ROM only needs to access to most significant bits of a floating point word, and so it appears on the upper datapath. There is also only one barrel shifter, which is a large macrocell. The barrel shifter is useful only for non-floating point data. The bus ties allow it to be shared by either datapath. However, using the barrel shifter from the upper datapath limits the amount of processing which can be done on the lower datapath on that clock cycle, since the lower and upper busses are tied together.

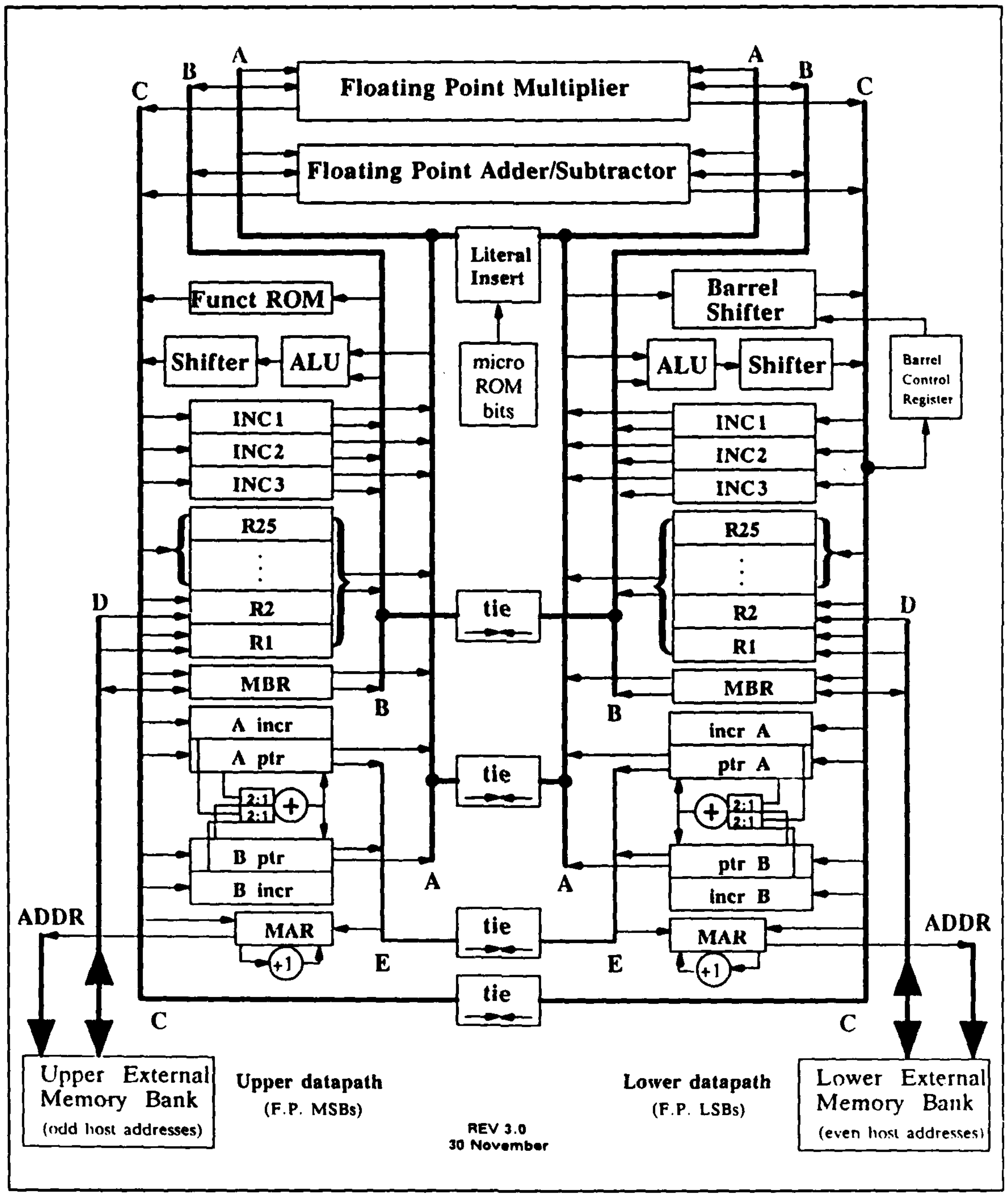

A
B
C
Floating Point Multiplier
Floating Point Adder/Subtractor
Literal Insert
micro ROM bits
Funct ROM
Shifter
ALU
Barrel Shifter
Barrel Control Register
INC1
INC2
INC3
R25
R2
R1
MBR
tie
D
A incr
A ptr
incr A
ptr A
2:1
+
B ptr
B incr
ptr B
incr B
MAR
+1
E
ADDR
Upper External Memory Bank (odd host addresses)
Lower External Memory Bank (even host addresses)
Upper datapath (F.P. MSBs)
Lower datapath (F.P. LSBs)
REV 3.0
30 November

Figure 3.3. FPASP Register-level Description.

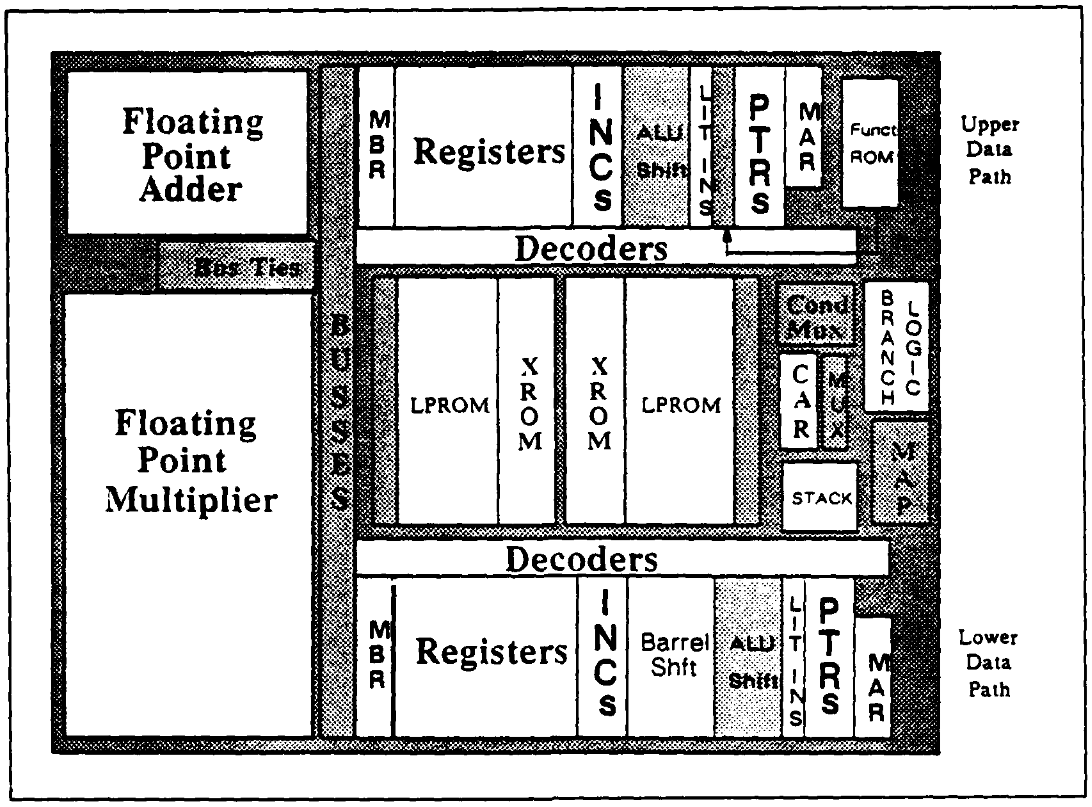


Figure 3.4. Basic FPASP Floorplan.

3.4 Bus Architecture

One architectural feature retained from the SPASP was the overlapping register set. This allows addresses or data to be stored in any register and moved to either the MAR or the processing hardware without going through an intermediate register. The overlapped structure is shown in the register-level description of the FPASP in Figure 3.3.

The **A** and **B** busses supply input data to the processing hardware, and results are returned on the **C** busses. The **A** bus can be driven by any register except the Memory Address Registers (MARs) which only drive out to pads. The **B** busses can only be driven by the general purpose data registers, the incrementable data registers, and the floating point processors. The **C** busses return data to any register, including the MARs.

The registers which are overlapped by the **A**, but not the **B** bus have their own "**E**" bus. The **E** busses carry only addresses, and so are only 20 bits wide. The registers which can drive the **E** bus must also drive the **A** bus, so they are all 32 bits wide. The exceptions to this are the MARs which only hold 20 bit addresses, never data. When the MARs load from the C bus they take in only the 20 least significant bits (LSBs).

This overlapped architecture can sometimes double the number of transfers that can occur in a single instruction. For example, data can be sent to the ALU on the **A** and **B** busses at the same time the pointer registers are sending addresses to the MARs via the **E** bus. The entire register set remains accessible via the **A** bus if the processors need data from the pointers on another clock cycle.

Another feature of the architecture allows both datapaths to share data or processing resources: the upper and lower busses can be tied to their counterpart. In this way, registers drive to or load from busses on the other half of the FPASP. The disadvantage of using the busses this way is that they are then 'tied up' doing one operation; the other datapath cannot use the tied busses for its own data transfers.

The ability to tie the **E** busses becomes most important when the FPASP is reading or writing floating point numbers. The most convenient way to store the 64 bit floating point numbers is to put them into the same address in each of the external memory banks. To get the same address into both MARs does not require that a pointer on each side hold the same information. The **E** busses can be tied together and driven from a single pointer register, or an address computed in the datapath of one side can be driven onto the tied **C** busses and loaded into both MARs.

Thus, the FPASP bus architecture supports three different configurations depending on how the busses are tied. Each of the datapaths can work separately on data in their own registers; or they can share data from each other's **A** and/or **B** busses and return results to either one on tied **C** busses; or the two datapaths can transfer floating point numbers to and from the external memory banks with only the **E** busses tied together.

The first version of the architecture contained only the **A** and **E** bus ties. Feedback from the microcode written by the students in the summer quarter "Introduction to Computer Architecture" (EE588) for a preliminary version of the FPASP pointed out the need for more interconnectivity to relieve dataflow bottlenecks, especially on the **A** bus. At first, the additional bus ties were not easily supportable in the hardware due to the space the busses would take up crossing the chip. A revised floorplan which put both floating point processing circuits on the left side of the chip forced such a crossover to be made; so adding the other bus ties no longer required additional space. The floorplanning of the chip is discussed in Chapter 4.

The only other bus on the chip which is accessible to more than one register is the **D** bus. This bus runs from the Memory Buffer Registers (MBRs) to the data pads. The **D** bus also goes to the R1 and R2 registers, which can only load from the bus. The additional registers on the **D** bus relieve an I/O bottleneck.

The **D** bus is an internal bidirectional bus which can be driven by MBRs for a data write to memory, or driven by the data pads for a memory read. The direction of data flow is controlled by the Write Enable Bar control bits. Each data path has its own control bit to the corresponding half of the external memory, so integer transfers can be independently controlled. A "1" in this control bit field puts the data pads in the input mode, which is the default value.

3.5 Register Types

3.5.1 **General Purpose Registers.** Five bits were allocated for controlling access to the **A**,**B** and **C** busses. This allows 32 choices for access to these busses. So in the case of the **A** bus: of the 32 possible choices, 1 is used for the NOP, 3 for the incrementable registers, two for the pointers and one for the memory buffer register, leaving 25 for general purpose registers.

The reason for using up all of the leftover choices on general purpose registers is the increase in efficiency provided by a large register set [Hen84] [Mit86]. This is because the processor does not lose a clock cycle getting the data through the memory buffer registers, and it does need to calculate all of the addresses for the data.

The choice of 5 bits for the register selection microword field is based on the size of the registers and the size of the decoders. The registers laid out for the ASP chip were used to floorplan the FPASP chip. Feedback from this floorplan indicated that more than 32 registers in each datapath would increase the size of the chip past the 350 square mil (0.35 sq. in.) limit set for cost and fabrication yield reasons.

3.5.2 **Incrementable Registers.** Loops are one of the common software structures found in the microcode written for matrix algebra on the SPASP, and also most other types of programs. The registers chosen to support this feature are incrementable data registers, each with its own half-adder circuitry and zero flag. It can increment without using any other hardware resources such as the busses or ALU. This is a critical feature, since it leaves those resources free for useful computation, decreasing the number of clock cycles needed in each loop.

To perform a loop, the user has only to load an incrementable register with the negative value of the number of loop iterations. On each iteration of the loop, the register is given the increment command and the flag is checked. On the last iteration, the incrementer has counted up to zero and the zero flag is set. To exit the loop, one of the instructions in the loop is a conditional branch based on the value of the incrementer's flag.

The FPASP has a total of six incrementable registers, three on each datapath. The original SPASP had only two registers, and mirroring that in the FPASP made it four. Some of the microcode written for the SPASP indicated that even four might not be enough, so an extra one was added to each datapath. Thus, the FPASP can directly support nested loops up to a depth of six. After that, the loop counter must be incremented, and the zero condition checked, by using the ALU INC operation.

3.5.3 **Pointer Registers.** The matrix addressing problem was also solved by using registers that had their own dedicated arithmetic circuitry. When a matrix is stored in row-major order, it is easy enough to retrieve the data in each row by incrementing an address pointer by one; but some operations require that the data be retrieved in a different order, such as reading a column of the matrix.

A simple increment-by-1 is no longer enough to produce the required address in one clock cycle. The FPASP provides a register to hold the increment amount and a dedicated adder to perform the increment. This keeps the pointer increment operation from taking up the ALU resources, as with the incrementable registers mentioned above. An increment holding register is provided for each pointer, and it can only load from the C bus.

The primary purpose of the pointer registers is to compute and hold addresses; but the overlapping bus architecture allows the pointer registers to communicate with both the data processing hardware and the address registers. Therefore, in addition to holding and calculating 20 bit addresses, the pointers can also hold and increment 32 bit data.

3.5.3.1 **Additional Increment Hardware.** Originally, the architecture called for a full adder for each pointer register. This is a case where feedback from both the hardware and software research worked to make the architecture more efficient. To perform a 32 bit addition in one clock cycle required a carry-select adder. SPICE simulations of the carry-select adder in the ALU showed that this operation could be done in 12 nanoseconds. This was fast enough to meet the one clock cycle requirement, but floorplanning indicated that such an adder would be too large to afford having one for each pointer register.

The architecture was altered so that the two pointers on each datapath would share a single adder, while each kept its own increment register. For the microcode written for the EE588 projects, not having both pointers incrementable on the same clock cycle did not increase the number of lines of code. The need to increment both pointers on the same clock cycle did not arise often, and when it did the second increment could be moved to another existing line of code.

In a continuation of this architectural style, the incrementable registers were also set up to share a single adder. Not only was this supposed to decrease the layout area, but it would also save a bit in the microcode word for each datapath, which had reached the 128 bit limit that could be supported by the GMAT CAD tool. Again, the microcode showed that multiple increments could be avoided without adding a line of code.

Spice simulation results from the ALU adder also indicated that the simpler circuitry of the incrementing adder would be fast enough even if the carry had to ripple down the entire 32 bits. A simple ripple adder incrementer is small enough to allow one for each incrementable register. Having the registers share a single incrementer would have added to the complexity of layout without saving very much space.

So each incrementer has its own adder, despite the software feedback, but the condensed control word was retained to meet the requirements of the GMAT tool. Since all of the register hardware exists on both the upper and lower datapaths, two increments can still be done in one clock cycle for both the incrementable registers and the pointer registers.

3.5.4 **Memory Buffer Registers.** The Memory Buffer Registers (MBRs) drive data to the pads via the D bus. They also were initially the only registers to read data in from the pads, but that architectural feature changed twice due to feedback from the software. The first change was made on the SPASP to allow the general purpose register R1 to also load data from the D bus to ease a dataflow bottleneck in the dot product routine. Data could then be read from the external memory into R1, and in the same instruction, results could be loaded into the MBR for writing out in the next clock cycle.

The role of the MBR was further spread out to include the R2 registers when a similar bottleneck arose in code written by Capt. Linderman for supporting complex arithmetic. Again there were conflicts using the MBR for incoming data as well as outgoing data. Now three registers have a direct interface with the memory. The MBR architecture changes caused by this software feedback were checked against the hardware layout to assure that they were easily implementable.

3.5.5 **Memory Address Registers.** Access to the external memory is controlled by one address register on each datapath. The Memory Address Registers (MARs) hold the addresses to be driven off of the chip. These two registers (upper and lower) can load addresses from three sources. The normal source is from the pointer registers via the E busses. The MAR can also load addresses from the 20 LSBs of the C bus, allowing any register to be used to hold an address. It can also load the incremented value of its present contents.

3.6 Processing Components

The FPASP is intended to support a variety of application specific algorithms without a change of hardware and without external co-processors. The choice of processing components included in the FPASP architecture reflects this approach to ASP design. The processing component set must be generic enough to meet the needs of a wide range of computations. The FPASP processing components support three types of computation: logic operations, 32 bit integer arithmetic, and 64 bit IEEE double precision floating point arithmetic. It can also perform parallel operations on shorter integers. For example, the 32 bit datapath can be organized as four 8 bit integers. This way four operations can be performed at once, as long as a carry out of one 8 bit integer operation does not affect the next most significant 8 bit integer. To avoid this overflow problem 16 bits can be used for each integer. Using this approach, the FPASP could perform 4 8 bit by 8 bit multiplies by using the 32 bit integer multiply option of the floating point multiplier.

3.6.1 **Integer Arithmetic and Logic Units.** The FPASP has two independent Arithmetic and Logic Units (ALUs), one on each datapath. This follows the basic mirror image architectural style applied to the register sets. These two units perform the logic and most of the 32 bit integer arithmetic operations. The one integer operation not performed by the ALUs is the 32 bit multiply, which is done by the floating point multiplier. The ALU and the linear shifter are shown in Figure 3.5.

The choice of ALU functions was initially based on the ASP library ALU cells. These, in turn, were based on the ALU functions of the basic processor in the Mano text [Man82]. The ASP ALU was extended for the FPASP to include the inverse logic functions NAND and NOR. This extension not only increased the variety of operations available, but also decreased the amount of decoding hardware needed to control the ALU. This is an instance where the hardware research made the architecture more efficient and more flexible.

The ALU functions supported by the FPASP are listed in Table 3.1. The FPASP can support operations on integers longer than 32 bits with the functions "ADd with Carry" (ADC) and "Subtract With Borrow" (SWB). These functions allow the present operation to factor in the carry or borrow generated by the previous operation. A borrow is the inverse of the carry out saved from the previous operation, so the same flip-flop can be used for both the saved carry or borrow.

Function	Value Passed to Shifter	Flags affected
		CARRY, OVERFLOW, SIGN, ZERO
MOVN	A	none
OR	A OR B	zero
AND	A AND B	zero
XOR	A XOR B	zero
MOV	A	zero
NAND	A NAND B	zero
NOR	A NOR B	zero
NOT	NOT A	zero
INC	A + 0 + 1	All Four
SET	A + B + 1	All Four, Sets CARRY
ADC	A + B + previous carry	All Four
ADD	A + B + 0	All Four
NEGA	$\overline{A}$ + 0 + 1	All Four
SUB	A + $\overline{B}$ + 1	All Four
SWB	A + $\overline{B}$ + previous borrow	All Four
DEC	A + 1 + 0	All Four

Table 3.1. ALU Operations.

The two ALUs generate separate flags which are stored in flip-flops, and can be used as branch conditions by the microsequencer. These flags are the ones available in most processors: the carry/borrow out, an overflow condition, a zero result condition, and the two's complement

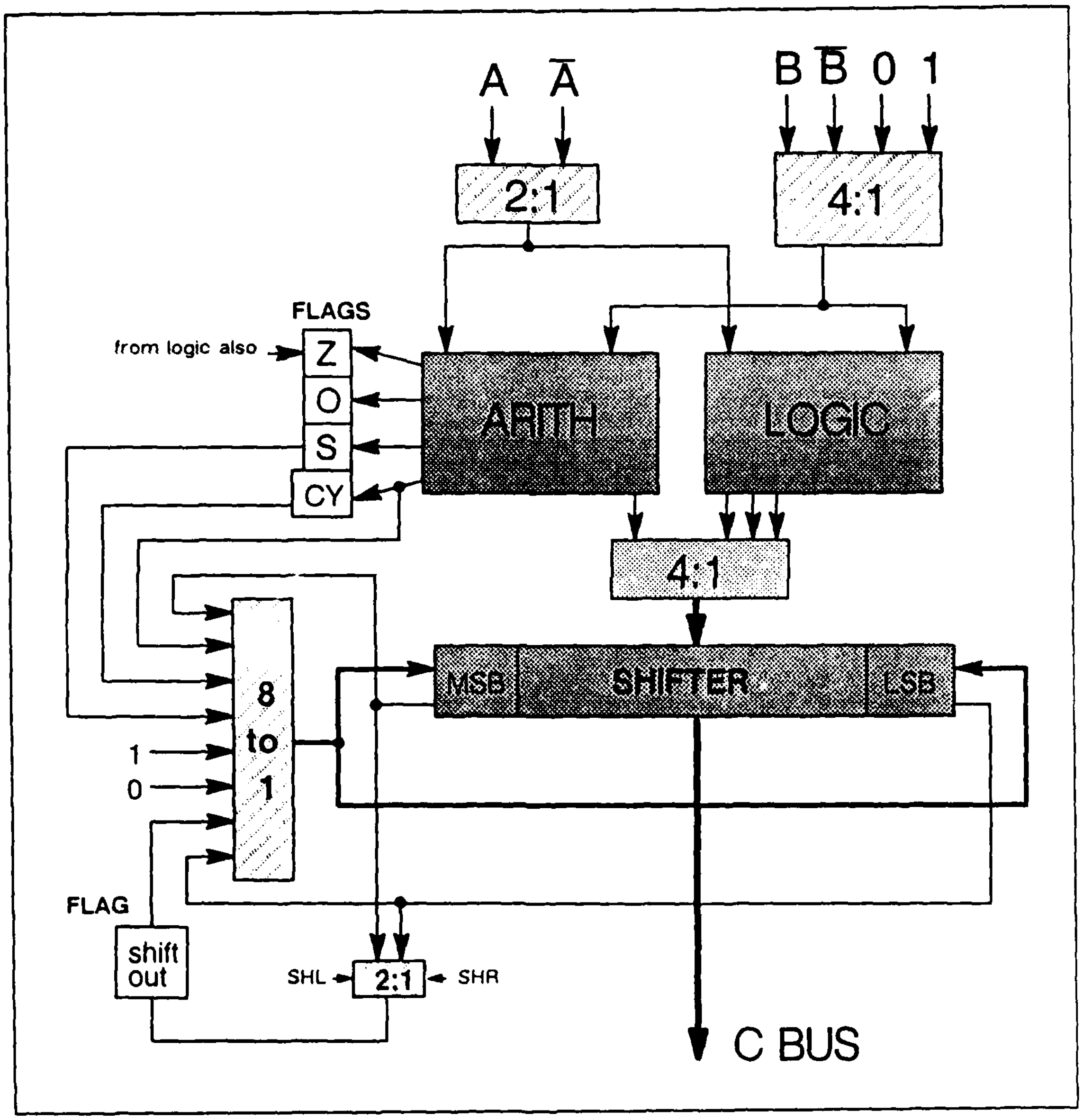


Figure 3.5. ALU and Linear Shifter.

sign of the result. The flip-flop adds a one clock cycle delay from when the flag is generated to when it can be used as a branch condition.

3.6.2 **Bidirectional Linear Shifters.** The result of the ALU operation feeds through a linear shifter before reaching the C bus. The linear shifter can perform a one bit shift in either direction. The shifts which can be performed are listed in Table 3.2. A block diagram of the linear shifter is shown in Figure 3.5.

The type of shift performed depends on the source of the bit shifted in, also shown in the table. The shifts include arithmetic shifts, logical shifts, circular shifts, and shifts using stored bits from the previous ALU operation. The bit shifted out is saved in a flip-flop, and it can be chosen as the shifted-in bit on the next clock cycle. The flip-flop for the shift out bit is loaded whenever a left or right shift is performed.

Function	Type of Shift	Bit Shifted In
NOP	Shifter does not drive C bus	none
GNDC	Shifter grounds C bus	none
PASS	No shift, ALU output goes on C bus	none
SLOT	Chained Left shift	previous shift-out bit into LSB
SLMS	Circular Left shift	MSB circulated into LSB
SLCY	Shift Left with Carry	Carry of present ALU operation
SL0	Shift Left with Zero	0 into LSB
SL1	Shift Left with One	1 into LSB
SRLS	Circular Right shift	LSB circulated into MSB
SRCF	Shift Right with previous Carry	Carry flag into MSB
SRS	Shift Right with previous Sign	Sign flag into MSB
SROT	Chained Right Shift	previous shift-out bit into MSB
SRSE	Arithmetic Right shift	MSB extended
SRCY	Shift Right with Carry	Carry of present ALU operation
SR0	Shift Right with Zero	0 into MSB
SR1	Shift Right with One	1 into MSB

Table 3.2. Shifter Functions.

The inclusion of a linear shifter reflects the goal of the FPASP's architectural specification process: support a wide range of operations. An example of how this process builds on itself was seen in the feedback from the EE588 projects. One of the groups used the linear shifter in their microcode and requested that the bit shifted out be made available as a branch condition. Hardware designed up to that point had not used up all of the available flag inputs, so the bits from the upper and lower shift-out flip-flops were added.

Another feature of the shifter allows simple initialization of a register to zero. The command GNDC causes the shifter to ground the C bus to all zeroes, which can then be loaded by any register. This leaves the A and B busses free to transfer data to the floating point hardware, or to the ALU for an operation that is only intended to set the flags.

3.6.3 **Barrel Shifter.** For shifts longer than one bit, a barrel shifter is included in the FPASP architecture. This device can perform a left circular shift of 1 to 31 bits in a single clock cycle. The barrel shifter takes its input from the **A** bus and puts the result on the C bus. The barrel shifter cannot pass the input straight through, the ALU/Shifter must be used for that.

Since the barrel shifter is only useful for integers, only one is provided. It has been placed on the lower datapath. This placement does not restrict its use to the lower datapath alone however; the bus ties described above allow the shifter to take in bits from the upper **A** bus and return the result to the upper C bus. This interconnectivity frees up space on the upper datapath for a different piece of hardware, increasing the flexibility of the FPASP architecture.

As with the linear shifter, the barrel shifter's flexibility was increased by feedback from the EE588 software projects. Use was made of all the FPASP features defined for those projects, and almost always the result was a need for an extension of those features. In the case of the barrel shifter it was the need to choose the length of the shift, based on a result in the datapath. The architectural solution was to allow a choice between the microword control bits or bits stored off of the lower C bus from any previous operation.

The barrel shifter can be made to perform a linear shift rather than a circular shift by setting the bits that will be shifted out of the left side to zero. This can be done by inserting a zero on top of those bits with the literal inserter, which is described in the next section. This is possible because the literal inserter drives the A bus, so it can affect the data being fed into the barrel shifter. Driving two values onto the A bus results in an AND operation, so by driving the register contents as well as the literal, the bits on the bus can be masked to zero.

3.6.4 **Literal Inserter.** A literal inserter is supplied in the FPASP to allow bits of the microcode word to be used as data. The literal insertion feature allows constants to be stored directly in the microcode rather external memory. This makes the FPASP more efficient in several ways: no clock cycles are wasted reading in the data or calculating its address, a memory location is not wasted on a constant, and the registers are not affected so data stored there need not be saved out to memory.

The size of the literal field in the microcode word is 16 bits, half the width of the datapath. This width is a compromise between the size of the microcode field and the width of the datapath. To supply a full 32 bits from the microcode would waste too many of the available bits. The maximum width of the microcode word is limited by the capabilities of the GMAT program and also by the size of the microcode ROM. Extra bits in the ROM add to its area and access time. These factors are covered in more detail in other sections.

The literal inserter places the data onto the 16 LSBs or MSBs of either **A** bus, where it is immediately available as data to the processing hardware. If the literal is placed on the lower bus LSBs it is at the input of the barrel shifter, which can then be used to rotate the bits to the MSBs, leaving the LSBs free to get another literal on the next clock cycle. This way a full 32 bit data word could be formed.

Here again, feedback from the Computer Architecture class pointed out the need to have the literal inserter drive the upper **A** bus also, despite the fact that the barrel shifter is not available there. The reason was that tying the A busses meant being unable to use the lower datapath, which would have added a clock cycle to the routine. In this, case the resulting hardware design was more complicated than the original literal inserter; but the increase in flexibility outweighed the increase in complexity.

3.6.5 **Function ROM.** The function ROM stores initial guesses (seeds) for iterative subroutines. These are routines which can calculate the square root or other function of a number in only a few clock cycles. An example would be a routine that uses the Newton-Raphson algorithm to calculate the square root or inverse of a number. The number of clock cycles needed to converge on the final result depends on getting a good initial guess, called a seed. The function ROM can store tables of precalculated seeds for eight different routines. The function ROM tables are partly predefined, for algorithms in permanent microcode memory. The rest of the ROM is laser programmable, so users can define their own seeds.

The function ROM is intended for use with floating point numbers. The table to be used for a particular function is selected by control bits. The MSBs of the floating point number's mantissa, and the LSB of the exponent are used to address a specific seed value. This seed provides the four mantissa MSBs and exponent LSB of the initial guess. The input bits are read off the upper **B** bus and the seed is placed on the upper **C** bus. Figure 3.2 shows which bits these are.

3.6.6 **Floating Point Multiplier.** The multiplier is the largest single cell in the floorplan. It has been designed and is being laid out by CPT Fretheim as a special project [Fre88].

The multiplier is a single combinational logic circuit which embodies the octal Booth's encoding scheme. The long combinational path requires that the multiplier be given two clock cycles to settle. This creates a pipelined microcode structure. If the multiplier is used continuously, it can put out results every two clock cycles. An extra cycle is needed to initially load the input registers. After that, the next input can be loaded in the same cycle that the result is driven out.

The selection of possible destinations for the result was determined by the microcode written for the dot product routine. Originally the multiplier was to drive only the C busses like the other data processing circuits. In the dot product routine, the adder accumulates the final result. It was more efficient to load the adder's previous result back into the adder's input register directly, rather than have the data be delayed by a register; so the adder was given the ability to drive the B busses also. The same choice of output busses was also given to the multiplier.

This permits 100% utilization of these two expensive resources. The clock cycle where the multiplier is settling can be used to load up the floating point adder, and vice versa, so the two are kept in continuous operation. This is done in the dot product routine, whose efficiency is a critical factor in the overall efficiency of the Kalman filtering routine. The dot product routine is listed in Appendix B1.

The multiplier supports the IEEE double precision floating point format. This requires that the multiplier flag the conditions of underflow, zero result, denormalized result, overflow, and a result that is not a number. These flags are available to the microsequencer so that branches to exception handling routines can be made. The flags which signal that an invalid result in any form has been produced are also OR'd together into a single flag called TRPS which can be used for a single condition check. This flag represents an overflow condition in the adder, multiplier, upper ALU or lower ALU, or the not-a-number condition in the adder or multiplier.

The multiplier is also designed to perform integer multiplies on 32 bit wide words, producing up to a 64 bit result. The LSBs of the result will be driven to the lower B or C busses. If the result is longer than 32 bits the extra MSBs are driven to the upper datapath and a flag is raised.

3.6.7 **Floating Point Adder/Subtractor.** The double precision floating point adder is being designed as an ongoing class project. It will also perform a floating point subtraction. The interface to the adder is identical to that of the multiplier. The adder will use a carry-select adder to perform the mantissa additions or subtractions.

The adder is also being given two clock cycles to settle, although it may be possible for it to settle in one clock cycle. That will not be determined until more Spice simulations are run.

3.7 Microsequencer

The FPASP is controlled by a microcode sequencer. The microcode is stored in two forms of ROM: a fixed store of microcode, and a write-once microcode store used to tailor the FPASP to the specific application. This architecture was extended to allow the microaddress stack to extend into the external memory, and to allow for external control of the FPASP. These two features were major areas of hardware research, and are discussed in greater detail in Chapter 4.

A diagram of the microsequencer is shown in Figure 3.6. The Control Address Register (CAR) supplies a 10 bit address to the microcode ROMs. The CAR can be reset to zero by the GO signal, so address 0000000000 is the start of the microcode routine. Ten bits allow up to 1024 words in the microcode ROM, but only 784 are used.

The CAR is loaded every clock cycle from a 4-to-1 multiplexer (mux). The choices of input include the incremented value of its present contents, an address from the microcode word presently on the control bus, the address on the top of the stack, or an address from the mapping ROM.

The incremented address is also the default selection, and is the selection used if the condition for a branch is false. The address from the control bus allows the program to branch to another section of the code, or to branch to a subroutine. If a subroutine branch is chosen, the incremented address value is pushed onto the stack. The choice of address from the stack represents a return from a subroutine call. The Mapping ROM holds addresses that are selected by bits in the lower R1 register. These addresses point to the support microcode for the assembly language.

Conditions for branching are selected from a 48-to-1 mux. Flags to one section of the mux can be inverted, giving a total of 64 branch conditions. Two of these selections are "unconditionally true" and "unconditionally false". The true choice is for branching without reference to a flag; and the false choice is the default, for executing sequential lines of microcode. The branch conditions and their sources are listed in the microcode field definitions in Appendix A1.

The choice of flags changed throughout the architectural specification process with feedback from the software and hardware research. One such change was mentioned above for the shifted-out bit. Another was a suggestion from the Kalman filter research group to have 8 externally settable flags. These were originally to be stored in a separate register, but this was changed to having them come directly from the upper R1 register's four MSBs and four LSBs. The idea of having the R1 flags input from outside the FPASP was intended only to allow the user to choose between separate routines in the ROM, a simplistic form of external instruction decoding. This feature was later extended to include input of control bits as well as flag bits.

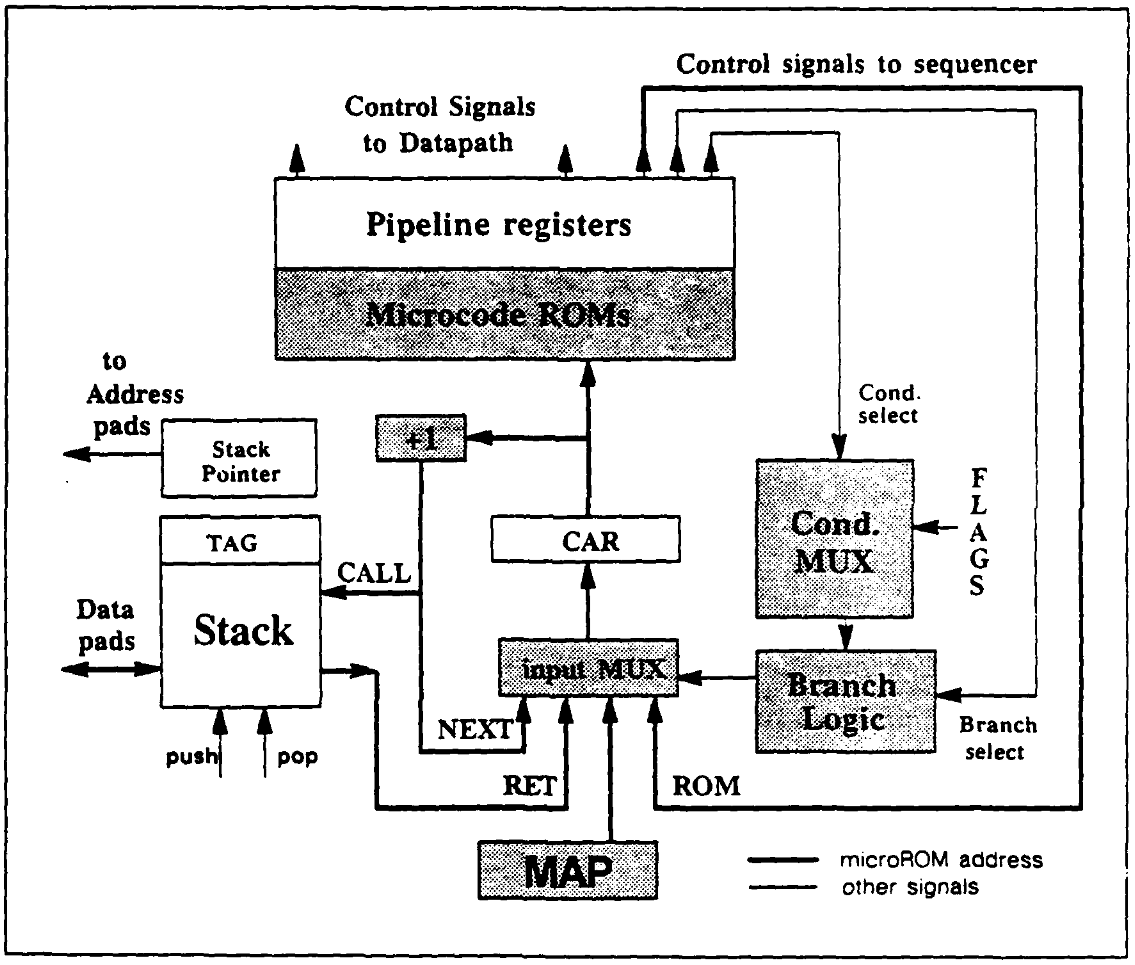


Figure 3.6. Microsequencer Block Diagram.

The branch condition and type of branch are selected by bits from the control bus and control bits generated in the microsequencer. The "branch logic" cell shown in Figure 3.6 is a collection of gates which use these control bits to select which of the four available addresses will pass through the 4-to-1 mux into the CAR. This cell also generates the control signals needed by the stack and MAP.

The selected address is loaded into the CAR and driven to the microcode ROM on the next clock cycle. The microcode control word out of the ROM is further delayed one clock cycle by a set of master-slave flip-flops called the Pipeline. This pipelined control architecture provides an orderly presentation of the control bits to the datapath, but it also introduces a latency between the time the address is chosen and the time its microcode control word is executed.

The results of this latency are that the flags cannot be used as branch conditions in the same line of code they are generated on, and the instruction immediately after the branch instruction is performed even if the branch is taken. These restrictions usually have no effect on the efficiency of the microcode, but sometimes a no-operation (nop) line of code must be inserted after a branch to prevent unwanted computation. Examples the effects of the latency are given in Chapter 4.

3.7.1 **Microaddress Stack Architecture for Recursion.** One of the original problems to be solved by the FPASP architecture was how to support recursive microcode routines. Since these routines continually call themselves, and can call other routines as well, the depth of the stack becomes the determining factor in the amount of recursion the FPASP can support. The stack registers take up area on the chip, so the depth of the stack cannot be increased without an area and cost penalty.

The solution chosen for the FPASP is to write the stack out to external memory when it gets full. After the stack is filled, each successive call pushes the lowest address on the stack out to external memory. A return then pops that address back into the bottom of the stack. The architecture of this stack is shown in Figure 3.7. For most applications the on-chip stack will be deep enough so that external memory will not be needed.

This architecture provides a pair of counters to keep track of the address and status of the external stack. The stack entry and address for the external memory are sent directly to the lower data and address pads, so the datapaths are not affected.

The only restriction this architecture places on the microcode is that there cannot be a lower memory access on the same cycle as the call or return. Except for this restriction, the operation of the stack is transparent to the user. Any inconsistency that arises from a stack operation causes

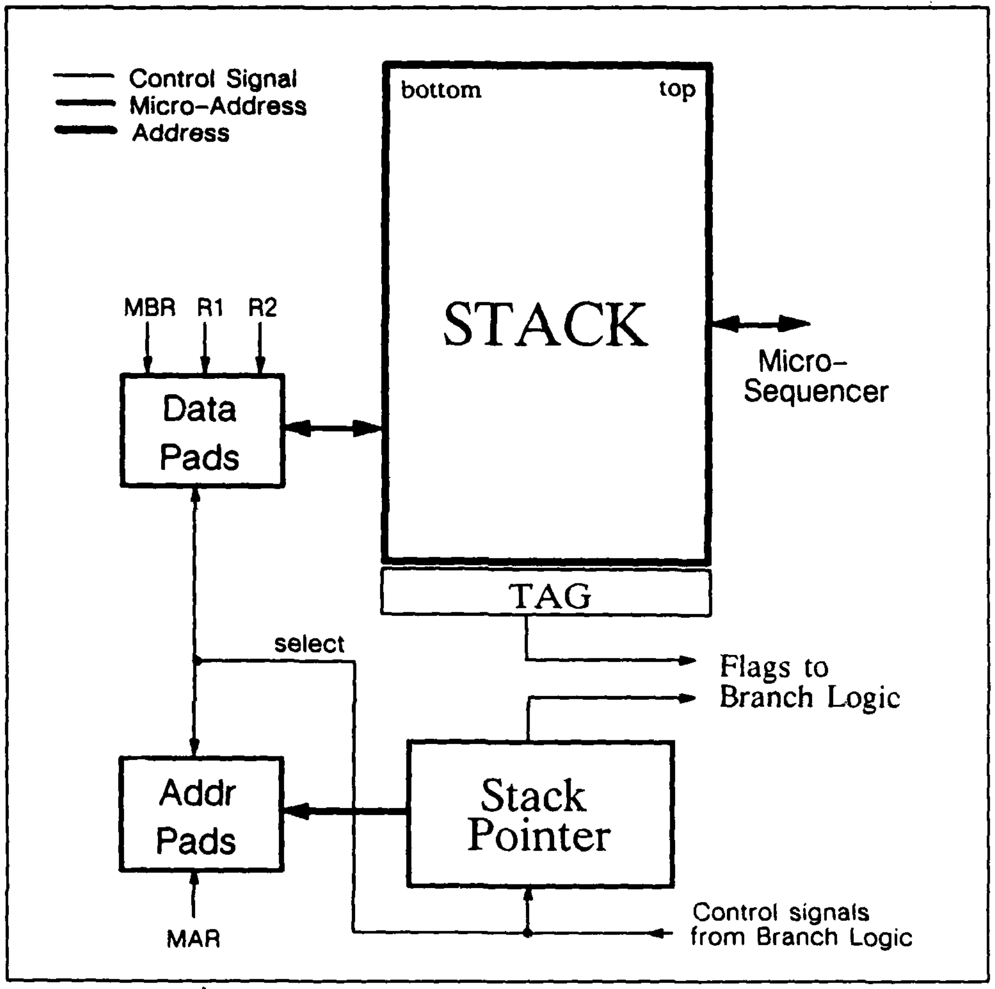


Figure 3.7. Stack Architecture.

the FPASP to branch to a trap routine which halts execution and saves the state of the entire machine. This routine uses the stack counters to perform the save. The programmer can use this routine also, or write one of their own if they do not want execution to halt on a trap condition.

3.7.2 **Microcode Storage.** The fixed portion of the microcode memory resides in the XROM. This is a standard cell compiled by software in the AFIT CAD environment. The program and cell designs are the result of AFIT thesis research [Ros85] and ongoing improvement by the AFIT faculty [Lin88-2].

The FPASP design methodology is to fabricate the FPASP in quantity and laser program it for each specific application. This approach to ASP design runs counter to the practice of designing a new ASP for each application. The technology that makes this possible is the Laser programmable ROM or LPROM. The laser programming technique is compatible with the MOSIS design rules and fabrication processes. If the designer had more control over the fabrication, a different programmable store could be used. A better one might be a UVPROM or the Flash memory recently announced by the Intel Corporation [Hit88].

The design of the LPROM is the topic of Capt. Tillie's research, which was carried out in parallel with this effort [Til88]. Capt. Tillie has written the tools needed to optimize the layout of the LPROM bits and drive the hardware which does the actual programming. The design for the LPROM cell was available at the time the FPASP was floorplanned. The LPROM is eight times less dense than the XROM, so there are fewer words of LPROM available.

The FPASP provides 640 words of XROM and 144 words of LPROM. Each word is 128 bits wide, but physically they are split into two 64 bit words. This split solves several problems. The first is that an XROM 128 bits wide would be too slow due to wide wordlines. The second is that it is easier to fit into the FPASP floorplan as two smaller pieces. The third is that the GMAT microcode assembly tool cannot handle integers larger than 128 bits at this time.

The split also made it natural to put control bits for the upper datapath on the upper ROM and likewise for the lower datapath control bits. Bits used for neither datapath were split up to balance out the number of bits on each half to 64. In the original floorplan of the FPASP, the ROMs were actually arranged facing their respective datapaths, reinforcing the notion of "upper" and "lower." This also decreased the length of the control busses. The ROMs are no longer in this orientation, but the bits have been rearranged so they are still as close as possible to their final destination.

Since there is less LPROM than the XROM, careful consideration must be given to which code belongs in the XROM and which will be written into the LPROM. The normal approach would be to write a routine into the LPROM which calls the generic routines in the XROM.

The FPASP must be able to do Kalman filtering even if the laser programming capability is not available; therefore the FPASP XROM contains all of the code needed to perform that algorithm. That routine has been written to use the matrix algebra routines that would exist in the XROM anyway, so the overhead of the more generic architecture of the FPASP is decreased.

3.7.3 **Microinstruction Pipeline.** The microcode control bits from the XROM or LPROM are loaded into the pipeline registers and driven out to the datapath hardware on the next clock cycle. This allows more time for decoding the control bits. The bits go out on the rise of the first clock pulse instead of after the precharge period and access delay of the ROMs.

This also isolates the control lines from the transients coming out of the ROMs. Since the ROMs are precharged, their outputs change to the precharge level and then settle to their correct value. This pulse of incorrect control would cause the decoders to dissipate power needlessly. It could also cause a change of state that could not be recovered from, as in the case of driving a precharged bus incorrectly.

The pipeline allows the control bits an entire clock cycle to settle before it latches them in. That feature becomes crucial when the control bits are to be overridden by bits from the datapath, which occurs in the case of the assembly language execution. The new control bits have plenty of time to arrive and stabilize. This way, their release to the decoders on the next clock cycle is synchronized just like ROM control bits.

3.7.3.1 **Pipeline Scanpath.** The introduction of the pipeline registers into the control path also makes it easy to add design-for-testing (dft) into the FPASP architecture. The purpose of dft is to decrease the amount of work needed to test the chip. The additional hardware gives the tester more controllability and observability over the state of the machine [Fuj85].

In the FPASP, the dft hardware consists of a set of muxes which allow the pipeline registers to be chained together to form a serial scanpath. This path allows the control bus bits to be shifted out to one pad at a time, providing complete observability of the control word. Controllability is also provided since the scanpath can also shift in new bits at the same time. Thus, the tester has more control over the state of the entire machine.

The clock lines to the pipeline can be isolated from the clock lines to the rest of the chip. The scanning is done while the clocks to the rest of the chip are disabled, so its state does not change with the changing control word. Separate test input and output pins allow the scanned word to be fed directly back in, so machine can pick up from where it left off.

3.8 Assembly Language Support

The idea of having flag bits from the upper R1 register was intended to allow the user to choose between separate routines written into the LPROM. The limited area available for microcode ROMs meant that the user written code was limited to 144 words.

This limit was approached in some of the code written for the EE588 class projects. Even though those routines were not fully optimized, the fact that they came close to the limit indicated that the FPASP should allow for external input of control words, not just flags.

So the FPASP architecture was extended to support an assembly language. Since the specification of the architecture was well along, the assembly language was designed to fit in with the existing hardware and to use a minimum of new hardware. Following this philosophy, only existing registers were used for the instruction register (IR) and program counter (PC). The upper R1 already had ties to the control section by the flags mentioned above, so it was used as the IR. The lower R1 also became necessary as an IR.

The way additional decoding hardware was minimized was to bring in identical copies of the control fields the ROMs put out. The original idea was to replace all the bits one-for-one, so there would only be a single instruction format. That was obviously impractical since it would require four registers and two memory accesses. It would also be wasteful since all of the internal capabilities are not needed when the code is supplied from the outside. For example, the literal inserter is not needed since the data would be coming in from the memory anyway.

One basic control to override is the register select fields, so the assembly instruction can specify the source and destination registers. The bus ties must also be overridden to get complete control of the entire bus architecture. These fields have a combined bit count of 33, larger than the upper R1, so the lower R1 was added. The **E** bus tie is not included in the count, since both R1's are needed to hold replacement control bits. The most efficient way to fill both R1's is to treat the assembly language words like floating point numbers and store both halves in corresponding memory locations.

The **A** pointer is used as the PC, so its increment register is loaded with a '1'. The microcode written to support this assembly language allows three addressing modes. The default mode is to use the incremented value of the **A** pointer. The 'immediate' mode takes the address in the 20

LSBs of the upper R1. The 'indirect' mode uses the address in any one of the four pointers to read in the address of the final address to be used. The final address in turn is put into the MAR.

The control pipeline latency does not affect the assembly language flow; all of that is absorbed by the support microcode. The flags are latched from the previous assembly instruction, and so are available immediately to the assembly instruction "branch."

There are six instruction formats. Each format has fields for the replacement control bits needed by the hardware used to execute that instruction. These fields are arranged to overlap as little as possible, so there are not too many muxes on any bit of the R1 registers.

When the replacement control bits are needed, they are selected by bits in a new microcode field. These new bits control muxes which select the IR bits over the bits coming out of the ROMs.

Since the actual control bits are replaced, most of the assembly language instructions can perform as many different operations with the FPASP hardware as the original microcode field. For example, the ALU/shifter instruction can do any combination of alu and shifter operations. On the other hand, the barrel shifter instruction cannot access the barrel control register since that mode of operations not feasible when several microinstructions are performed for each assembly language instruction.

The increased number of clock cycles needed for the assembly language instruction makes that method of programming the FPASP inefficient for time critical operations such as the dot product routine, which is needed N cubed times in a matrix multiply. Rather, the assembly language should be used to extend the sequential, routine-calling portion of the application. Thus, the heavy processing is done by the more efficient microcode routines in the XROM library or those made up by the user in the LPROM.

One piece of additional hardware that could not be easily avoided was a decoder to select the address of the microcode for the operation the opcode represents. Since a small LPROM already existed in the form of the function ROM, the mapping cell was made an LPROM. The decision to do the MAP with an LPROM gives this assembly language an additional feature: it can be partially user-defined.

The MAP is addressed by the five opcode bits in the lower R1. Five bits allows 32 opcodes. Of the 32 possible opcodes, one is used for the TRAP routine, ten are used for the pre-defined instructions, and 21 are left over for the user to program. The predefined macro instructions are listed in Table 3.3 along with the operations they can perform.

Instruction	FPASP Operations
Load	load one or two registers from external memory
Store	store one or two register's contents to external memory
Branch	branch to an external memory location on any branch condition, using direct or indirect addressing
Call	call an external subroutine, using direct or indirect addressing
Return	return from an external subroutine
ALU	perform any ALU operation with one or both ALUs with any source or destination registers
Bshift	perform a barrel shift with any source or destination registers
Ptr/Inc	increment a pointer or incrementable register, or load a pointer's increment register
FP+	perform a floating point addition, with any source or destination registers
FP*	perform a floating point multiplication, with any source or destination registers

Table 3.3. Assembly Language Macro Instructions.

3.8.1 **A User Defined Assembly Language.** With laser programmability in the microcode LPROM and the mapping LPROM, the user can make up the code needed to support an assembly instruction, and then assign the address of that code to an unused opcode and enter it into the MAP. The control bits used for the pre-defined instructions are also available to the user and they cover almost all of the hardware in the FPASP. The user can override those fields at any time in their own routine, but is restricted to the six choices listed in the microcode field definition in Appendix A1.

The most common use would probably just be to branch to a specific routine in the microcode depending on the opcode entered, but instructions useful to a specific assembly language program could also be created. For example, the support code for performing a multiply-and-accumulate could be written into the LPROM, with a branch at the end to the FETCH routine used by the predefined assembly language routines. The multiply-and-accumulate would then be a new instruction in the assembly language. The control bits for selecting the source and destination registers for the new instruction would be selected at the start and end, just like the predefined floating point assembly language instructions.

3.9 External Interfaces

The FPASP depends on external circuitry to store data, and to tell it when to start processing. Two different memory chips were researched for the FPASP. These were static RAM chips with access times less than 35 nanoseconds. These criteria were based on the need to perform memory accesses in one clock cycle, and the desire to eliminate refresh circuitry and timing problems associated with dynamic RAMs.

The external memory would be best organized with each memory chip providing one bit of the data. This memory organization simplifies addressing since no chip select lines must be decoded from the address bits put out by the FPASP. If such a decoder were required in the future, it would be best located on the FPASP so that the access time does not suffer from having external circuitry in the address path.

The researched memory chips only require two control signals, and they have a zero nanosecond delay from when the data becomes valid to when the write enable signal can rise [Per88]. This simplifies the timing of the signals from the FPASP. The three signals in the microcode from the original chips were retained. The extra bits would allow the FPASP to control other external circuitry if required.

The FPASP also communicates with the host processor. For its first application, the FPASP will be mounted on a board which is plugged into the VMEbus of a Sun workstation. The general layout of this board is shown in Figure 3.8. In this case, the host communicates with the FPASP through the board's interface chip.

The entire operating cycle of the circuit would follow the timing diagram shown in Figure 3.9. The host fills the external memory with data, puts its address and data lines to the memory chips in a high impedance state, then raises the GO line. When the FPASP finishes it switches its bus drivers to high impedance and raises the DONE line. This signals the host that it can unload the results and start over again.

The FPASP also has an external interface in the form of two pins which run directly to the condition mux in the microsequencer. These allow the external circuitry to control the flow of the microcode. These flags are called IO1 and IO2. They could be used for interrupt signals if the FPASP controls external devices.

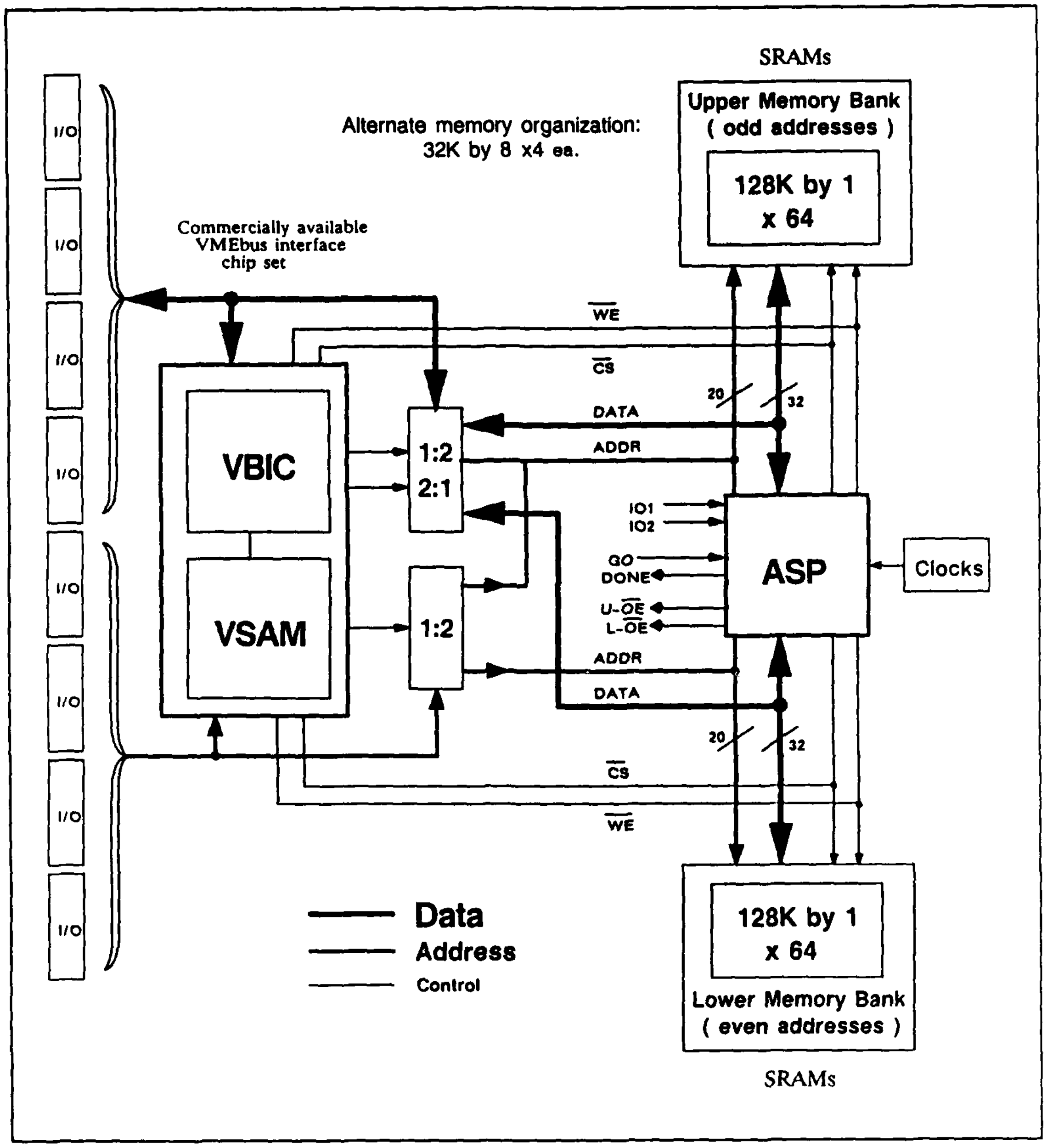


Figure 3.8. Circuit Board for VMEbus.

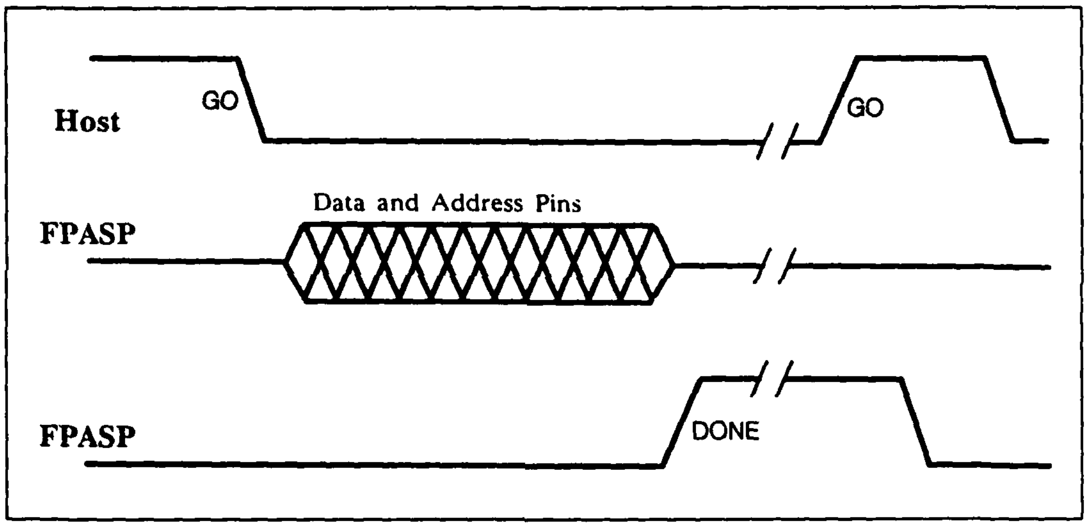


Figure 3.9. Host-FPASP Handshaking.

IV. VLSI Implementation

4.1 Introduction

This chapter presents the hardware designs that will embody the FPASP architecture. This corresponds roughly to VLSI portion of the design hierarchy shown in Figure 4.1. The implementation will be in CMOS, so that influence cannot be completely left out of the picture.

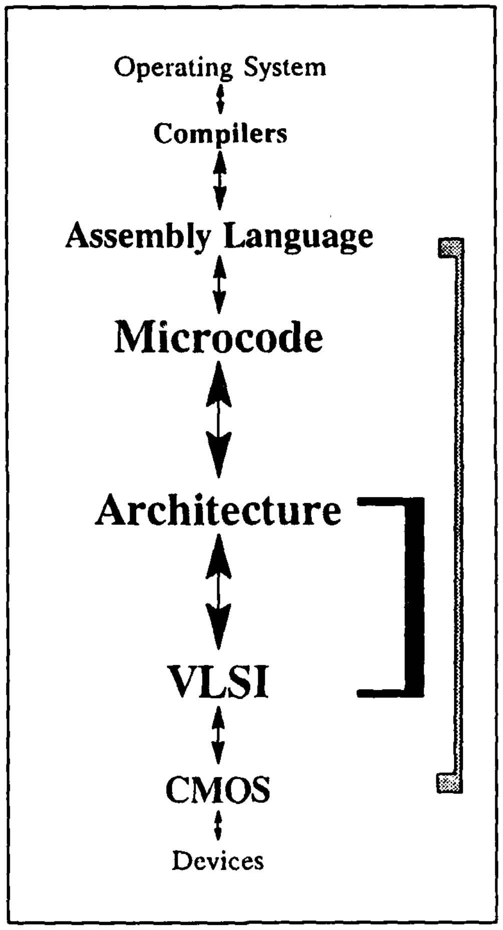


Figure 4.1. Design Areas Covered in this Chapter.

The discussion will be mostly at the block diagram level of detail, though occasionally the details of the CMOS design will be necessary to fully explain why a particular design was chosen. Microcode examples will be used to show how the hardware has been designed to support software structures.

4.2 Floorplanning

The hardware design began with a survey of the cell libraries available, especially the ASP library created by Capt. Gallagher. Even though most of these cells could not be used in the FPASP directly, they did provide a good size estimate for the FPASP floorplan.

After the first register level description of the FPASP was done, outlines of the hardware it called for were laid out in magic as large squares of different layers. Different layers were used only for their colors, so smaller macrocells would be easy to see. Contact layers were not used since they do not plot as blocks; they plot as a mass of small dots.

These blocks of color were then arranged in as square a fashion as possible, while maintaining what was felt to be a good organization. "Good" organization was one that limited the length of the various busses, especially the control busses. A drawing of this original floorplan is shown in Figure 4.2.

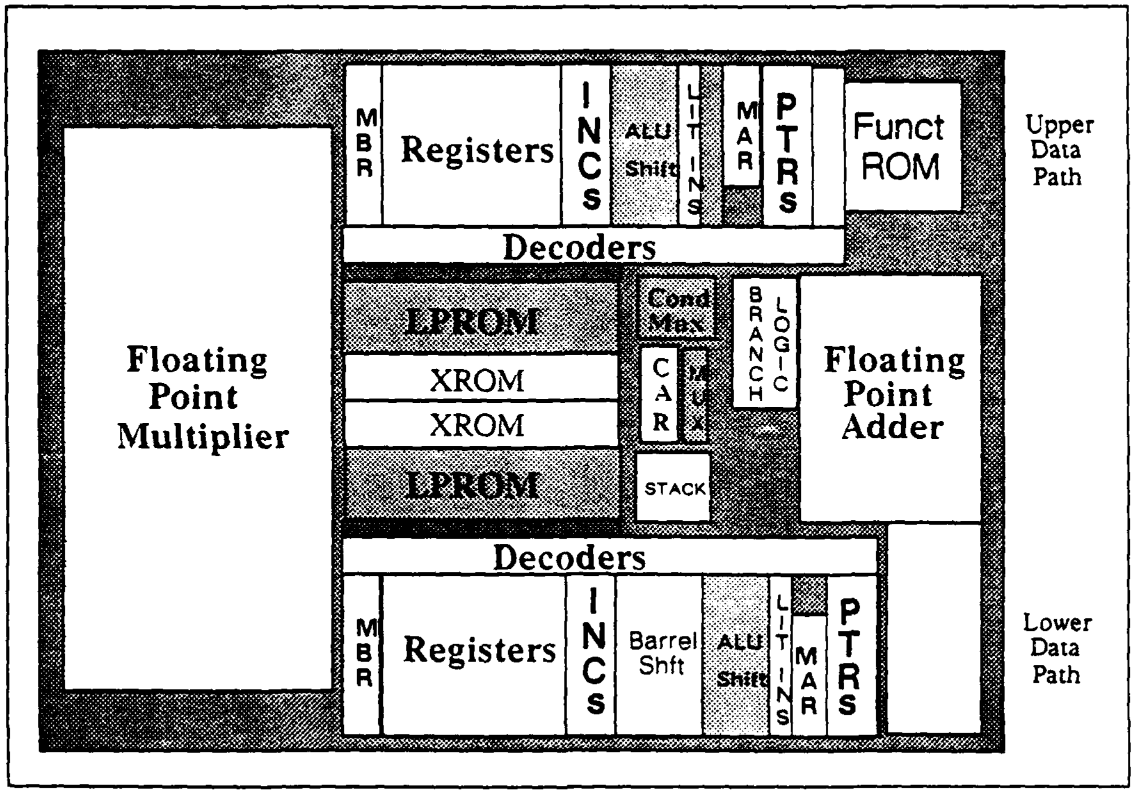


Figure 4.2. Original FPASP Floorplan.

The estimates of the datapath hardware were accurate, but only very rough estimates of the floating point hardware were available. At first, the adder was given a large area and put on the opposite side of the floorplan from the multiplier. This turned out to be impossible to build because the same physical channel was used for the **B** bus of the data registers and the **E** bus of the address registers to create the overlapped register sets. This isolated the adder from the **B** bus.

When a better estimate of the size of the floating point multiplier and adder were available, the floorplan was rearranged, with the result that the adder and multiplier were placed side-by-side. The final floorplan is shown in Figure 4.3. The rearrangement of cells had a great effect on the architecture as was seen in Chapter 3, with the addition of the **B** and **C** bus ties. What made the extra bus ties possible was that the busses all came together between the floating point macrocells. The extra ties no longer required additional channel space since the extra channel was already paid for to get all of the busses to the adder.

Better estimates also became available for the microcode ROMs. The new arrangement allowed the ROMs to be rotated and made deeper since they were no longer affecting a critical dimension of the chip. Another result of the new floorplan was that the control section was put next to its I/O pads, rather than having signals go around or through the adder. This can be seen in the pad arrangement indicated in Figure 4.3.

4.3 Timing

The FPASP is designed for a clock cycle of 40 nsec, corresponding to an operating frequency of 25 MHz. The clock cycle is shown in Figure 4.4. The clock signals are non-overlapping to prevent data from racing through the master- slave flip-flops. The Φ_1 clock pulse is shorter than the Φ_2 clock because it is also used as the precharge signal. The Φ_2 clock pulse width is less critical, it only needs to be non-overlapping and long enough for the registers to latch their inputs. The actual pulse width used for Φ_2 can be whatever meets these criteria and is easiest for the clock generator to produce.

The FPASP has two pins for the precharge signal even though the design calls for precharge and Φ_1 to be the same. This lowers the load on the Φ_1 pins used for logic, and allows precharge to be separated from Φ_1 for testing. The clock lines to the pipeline registers also have separate pins so that they can be isolated from the rest of the circuit for operating the serial scanpath for testing.

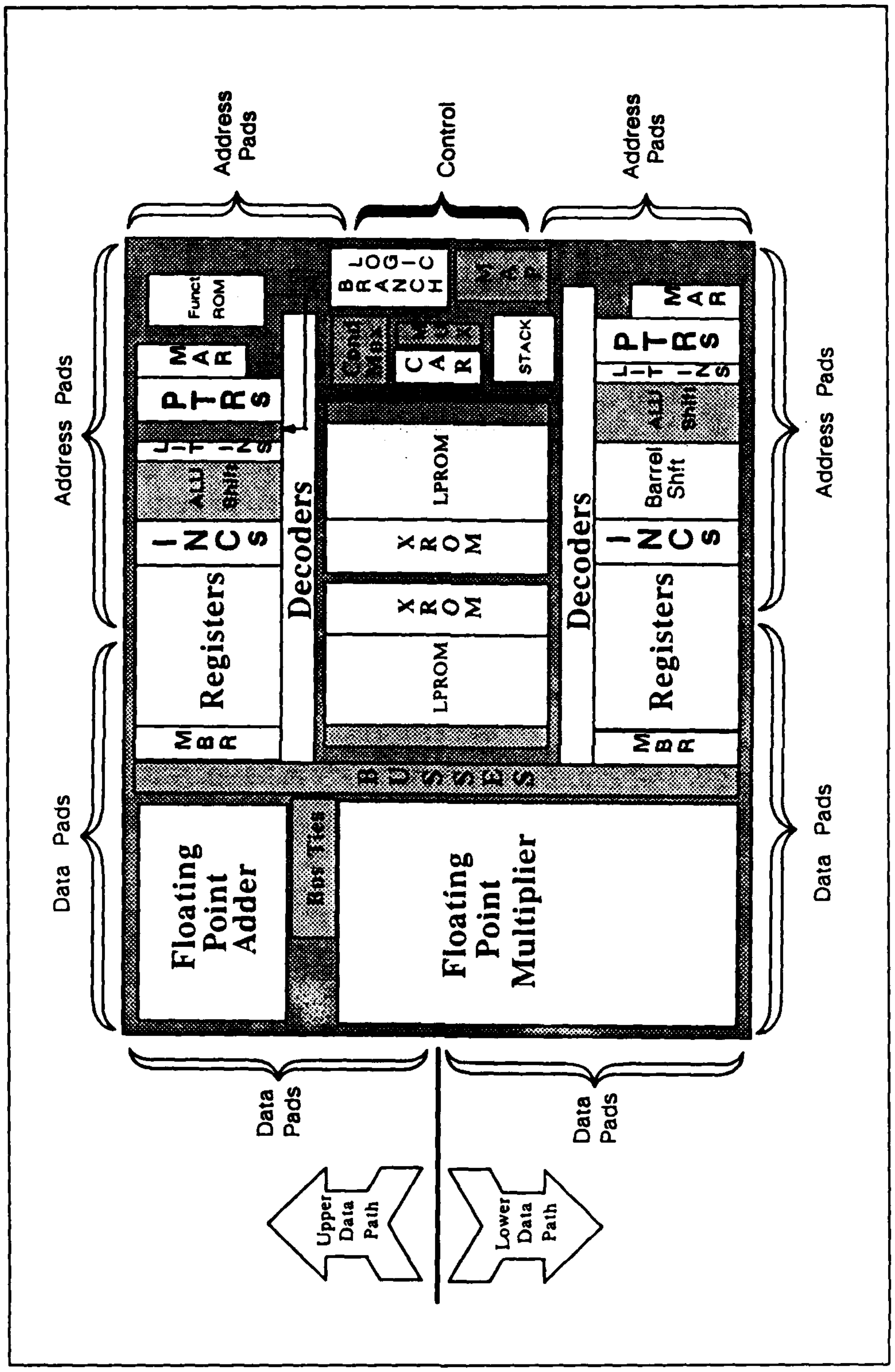


Figure 4.3. Final FPASP Floorplan.

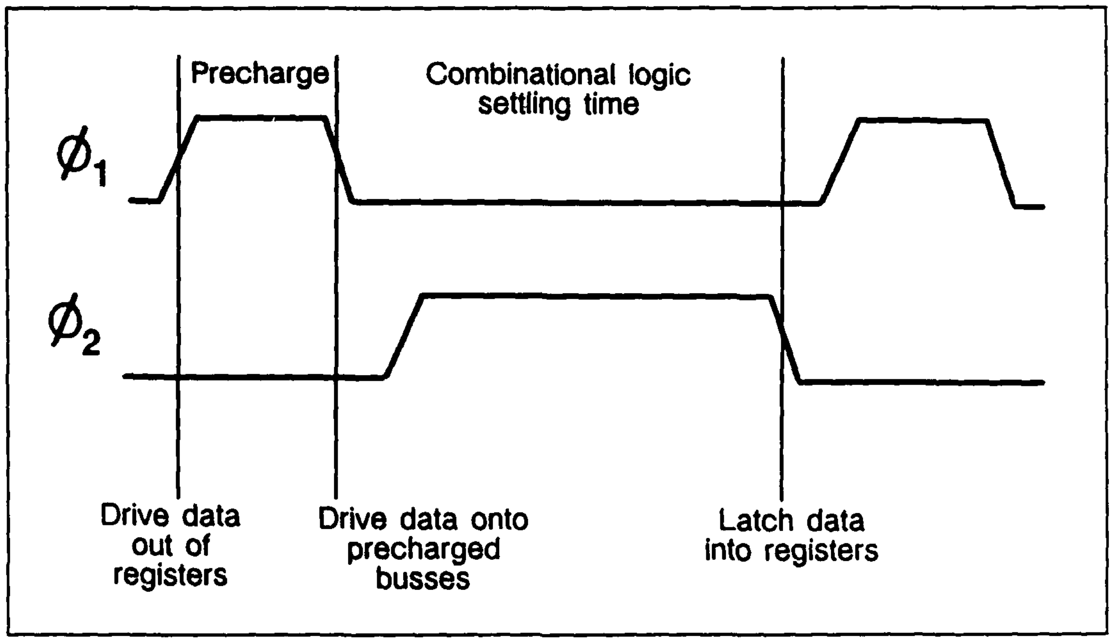

Precharge
Combinational logic settling time
ϕ_1
ϕ_2
Drive data out of registers
Drive data onto precharged busses
Latch data into registers

Figure 4.4. FPASP Clock Cycle.

4.4 Busses and Ties

The FPASP has a large set of busses for the various operations it performs. The arrangement of the busses inside the various cells is shown in Figure 4.5. This bit-slice arrangement was the main design feature carried over from the ASP cells. The arrangement of the various busses and the possible ways to interconnect them are shown in Figure 4.6.

The connection of the upper and lower halves of each bus are done with the bus ties. Interconnection between the **A** or **B** busses and the **C** busses can take place through the ALU/Shifters using the MOVN,PASS command. Data can be swapped between upper and lower registers in a single clock cycle using these commands and the bus ties; an unplanned but useful result of adding the extra bus ties.

The bus ties are large pass transistors. The control for each tie is a single bit in the microword that drives the transistors' gates. For the precharged busses, a single N device suffices, while the driven busses require a full T-gate.

4.4.1 **Precharged Busses.** The FPASP uses precharged **A**, **B** and **E** busses. This design was chosen for several reasons, the first being that the ASP cells were designed for these busses.

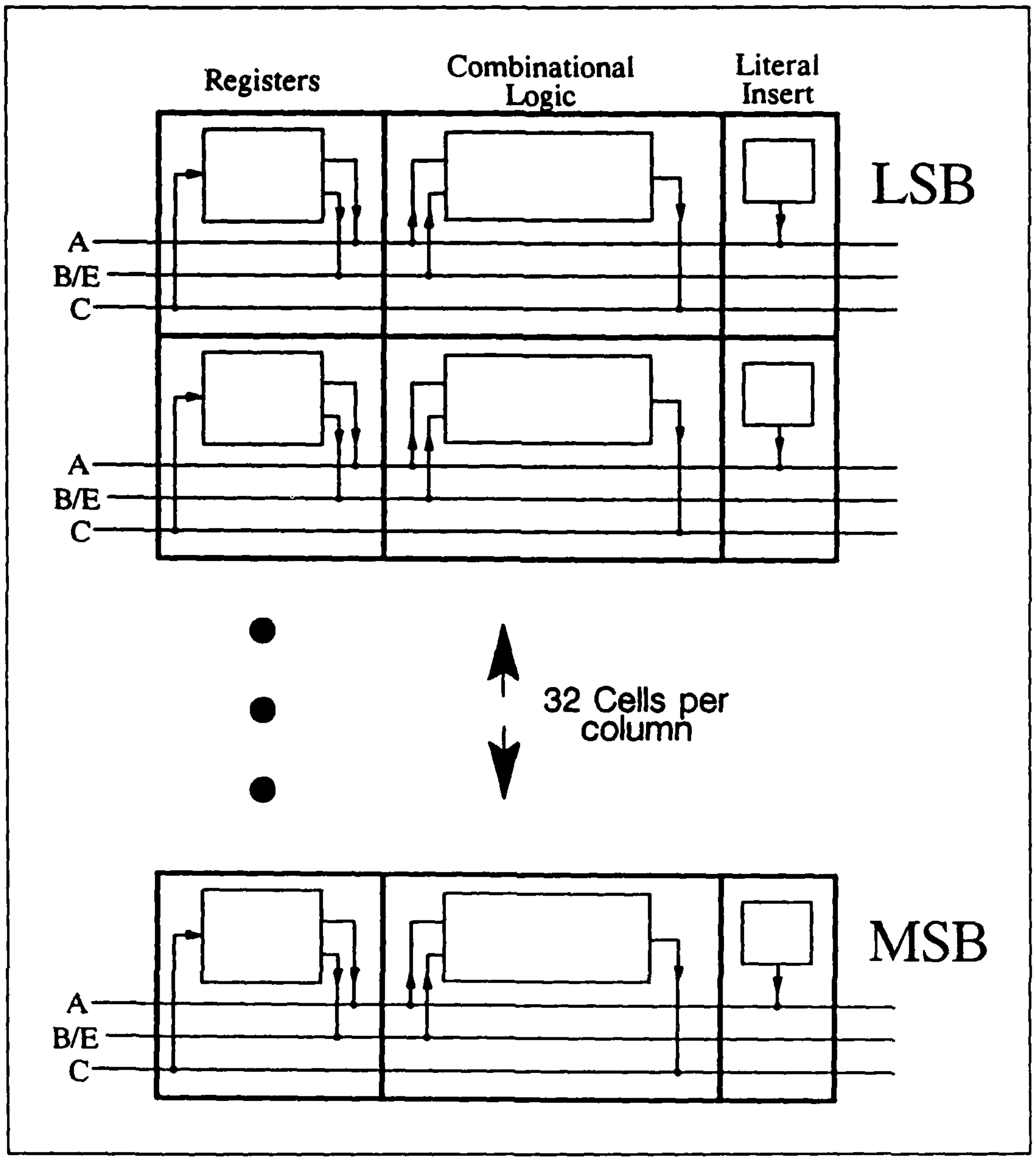


Figure 4.5. Bit-slice Bus Architecture.

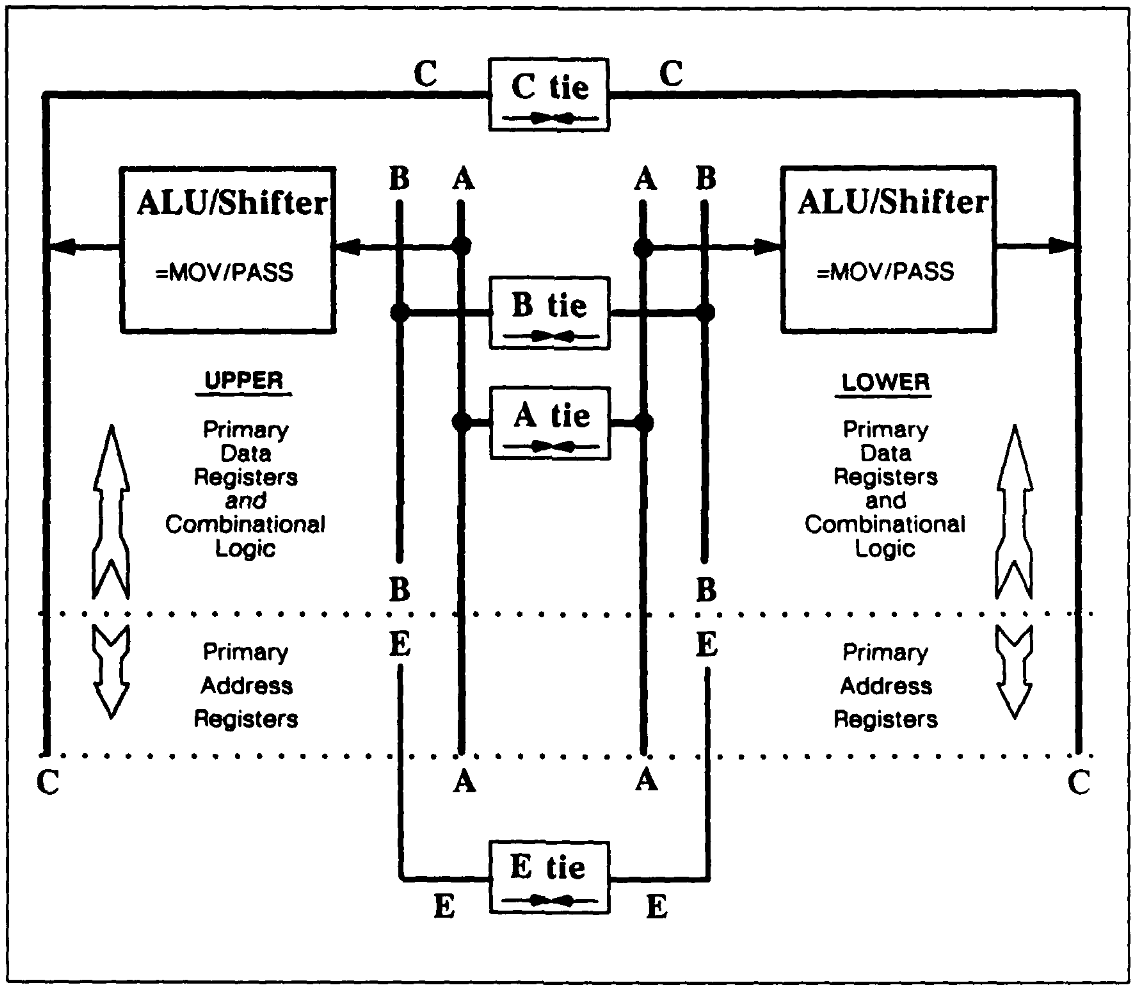

C
C tie
C
ALU/Shifter
=MOV/PASS
B A
A B
ALU/Shifter
=MOV/PASS
B tie
A tie
UPPER
Primary Data Registers *and* Combinational Logic
LOWER
Primary Data Registers and Combinational Logic
B
B
E
E
Primary Address Registers
Primary Address Registers
C
A
A
C
E tie
E
E

Figure 4.6. Bus Interconnections.

Precharged busses offer some advantages over a driven busses. The most important one is speed; there are over 30 drivers on these lines, each with some drain capacitance. Eliminating the larger P devices from the bus removes more than half the drain capacitance. The result is less charge that has to be moved on and off the bus to change its state. This speeds up the transition and decreases the rise and fall times. Shorter rise and fall times allow the inverters gated by the bus to dissipate less current since they pass through their switchover point faster.

Removing the P transistors also makes the register cells much smaller since the P transistor is usually the larger of the two drivers. The details of sizing the transistors are discussed in more detail in Chapter 5. With the P device out of the cell, the decoder no longer has to supply the inverse of the drive signal.

4.4.2 **Driven Busses.** The **C** busses are driven because the macrocells that drive them have asynchronous settling times, and the bits driven out may switch as the circuit settles. Most of the load on the **C** busses are the drains of small T-gate muxes. The drivers for these busses do not need to be as large as the ones on the A and B busses since the total drain capacitance is small.

The busses from the MBRs to the pads are called the **D** busses. These busses are also accessible to the R1 and R2 registers for reading in data. They are driven since there is also little drain capacitance on these busses, making them easy to drive quickly. The same is true of the Address busses, which go only from the MARs to address pads.

4.4.3 **Bus Select Decoders.** The signals that control access to the A,B, and C busses come from a common set of decoders located below the registers. Each control field is five bits wide, allowing for 32 choices. The choices for each field are listed in the definitions of the microcode fields listed in Appendix A.

In all cases the default code of 00000 is used for choosing no macrocell to load from or drive onto a bus. This allows cells not controlled by these fields to have access to the busses. The *microcode must not cause a condition where two macrocells* are driving a bus at the same time. An exception to this rule involves the Literal Inserter, which can be used to mask data on the **A** bus by driving the bus at the same time as a register.

Decoding circuitry to prevent this control problem would be complex and take up extra area for logic and wiring. It would also slow down the control signals, possibly preventing them from settling fast enough. Therefore, deconflicting of the microcode is left up to the programmer.

The decoding circuitry that controls connections with a precharged bus must contain a gate for enabling the output only when the precharge cycle is over. This prevents the selected register's pulldown device from fighting with the pullup and dissipating power as heat. Also, since the bus has no way to recover its charge, this gate is always the last combinational gate before the stage-up and driver inverters. This way, a straggling control bit will not change the state of the control line and cause a bus to be discharged in error.

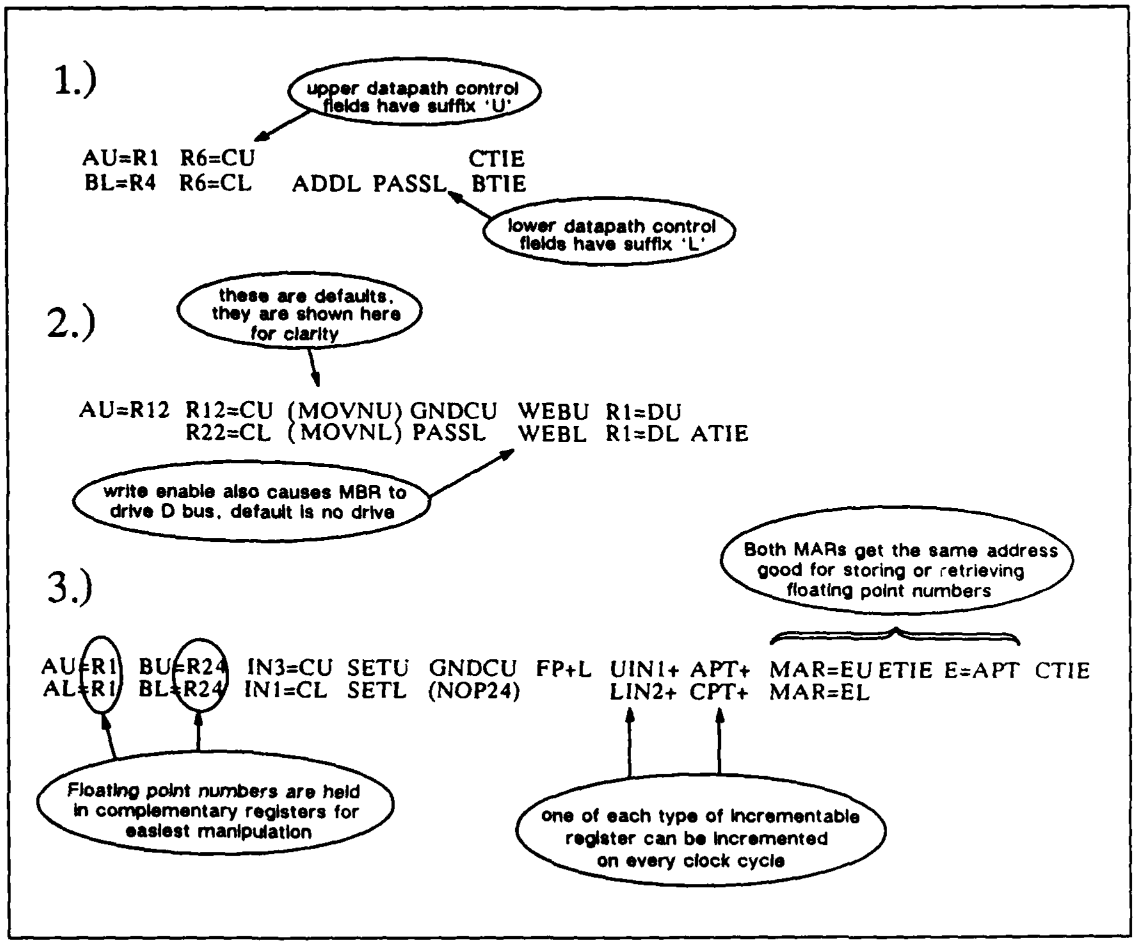
In addition to the three general bus select fields, the special purpose registers have their own control fields. The MAR control field choices NMARU and NMARL take the FPASP off of the external address busses by putting the address and data pads in a high impedance output state. This is the choice used when the FPASP is done processing so the host can take over the external busses.

The MBR control field controls the registers which load from the **D** bus: the MBR, R1, and R2. This makes possible another illegal microcode combination: loading R1 or R2 from the C and D busses at the same time. R1 and R2 are also treated as general purpose registers and so they have the same decoders as those registers, in addition to the **D** bus decoder. The MBR drives the **D** bus only when the Write Enable bit for that datapath is low, which causes a memory write.

4.4.4 **Microcode Examples.** This section will present portions of FPASP microcode that demonstrate how the bus design make possible efficient routines. Figure 4.7 shows three examples. The first example shows the lower ALU adding inputs from both the upper and lower datapaths, and returning the sum to both datapaths.

The second example performs a set of data moves using the **A**,**B**,**C** and **D** busses. Data is written out of the MBRs and also saved over in the R1's. Meanwhile, the upper R12 is cleared, and its previous contents are passed to the lower R22 through the tied **A** bus. Note that the **E** busses cannot be used for data, since they go only to the MARs.

The third example shows a line of code which uses every bus on the FPASP. The **A** and **B** busses are loading data into the floating point adder, while the tied **C** busses are being grounded by the upper Shifter to reset two of the incrementable registers. The E busses are tied together so the A pointer can be loaded into both MARs. Meanwhile, the MARs and MBRs are writing out the result of a previous operation, using the Address and **D** busses. At the same time, the ALUs are setting their carry flags for subsequent operations or branches. While all that is going on, the A and C pointers are being incremented, as well as two incrementable registers.



```
1.)

AU=R1  R6=CU                    CTIE
BL=R4  R6=CL    ADDL PASSL      BTIE

2.)

AU=R12  R12=CU  (MOVNU) GNDCU  WEBU  R1=DU
        R22=CL  (MOVNL) PASSL  WEBL  R1=DL  ATIE

3.)

AU=R1  BU=R24  IN3=CU  SETU  GNDCU    FP+L  UIN1+  APT+  MAR=EU ETIE  E=APT  CTIE
AL=R1  BL=R24  IN1=CL  SETL  (NOP24)        LIN2+  CPT+  MAR=EL
```

upper datapath control fields have suffix 'U'
lower datapath control fields have suffix 'L'
these are defaults, they are shown here for clarity
write enable also causes MBR to drive D bus, default is no drive
Both MARs get the same address good for storing or retrieving floating point numbers
Floating point numbers are held in complementary registers for easiest manipulation
one of each type of incrementable register can be incremented on every clock cycle

Figure 4.7. Microcode Examples of Bus Usage.

4.5 Register Arrays

The registers available to each datapath are shown in Figure 4.8. The decoders face the center of the chip, towards the control section. The split between the **B** and **E** busses is at the edge of the pointers. To the left of the split is all of the processing hardware, and to the right are the pointers and address registers.

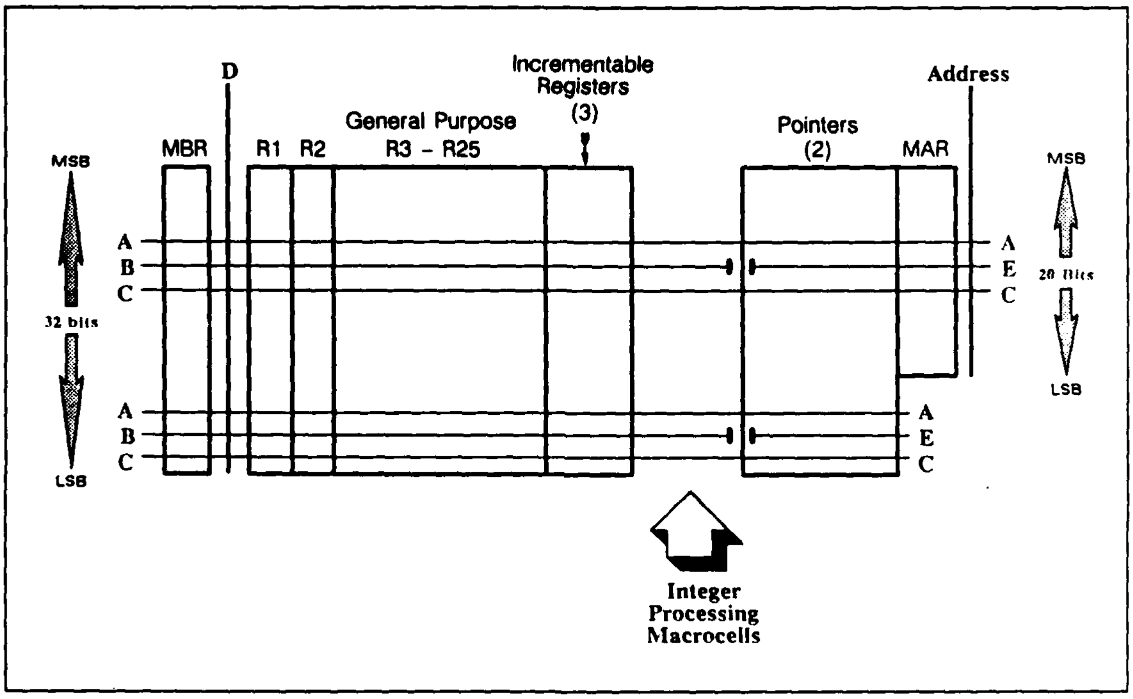


Figure 4.8. FPASP Register Set.

All of the registers are master-slave flip-flops. The input is latched on the fall of Φ_2. The outputs are driven on the rise of Φ_1 for the driven busses, and the fall of precharge for the precharged busses. This type of register can be loading in a new value at the same time it is driving out the old one.

4.5.1 **Additional Increment Hardware.** The six incrementable registers (three on each datapath) are provided to support looping in the microcode. This is done by having the incremented version of their present value available for loading on the next clock cycle. They also produce a flag to indicate that they have loaded a zero.

These registers will produce the zero flag whether they have loaded all zeroes from either the C bus or the incrementer. So they can be used to generate a zero branch condition if the ALU is busy doing something else. They can also be used as general purpose registers if needed.

The incrementer logic is derived from the logic for a full adder whose B input is always zero:

$$\text{Sum} = A \oplus 0 \oplus \text{Cin} = A \oplus \text{Cin}$$

$$\text{Carry out} = A \cdot 0 + A \cdot \text{Cin} + 0 \cdot \text{Cin} = A \cdot \text{Cin}$$

For the first stage, Cin (carry in) is always 1, so the logic further reduces to:

$$\text{Sum} = A \oplus 1 = \overline{A}$$

$$\text{Carry out} = A \cdot 1 = A$$

The incrementable registers use a ripple half-adder to generate the increment of the register value. The 'A' input to this incrementer comes directly from the register, so it starts to change as soon as the slave half of the register loads. This allows nearly the entire clock cycle to propagate the carry through the adder.

The data in the incrementable registers can be incremented every clock cycle, but the incremented value of newly loaded data is not available on the next clock cycle. It takes one cycle to get the value through the incrementer so it is not available until the second cycle after it has been loaded.

4.5.2 **Pointer Registers.** The Pointer registers actually consist of three parts: the register which holds the pointer value, a register to hold the increment amount, and a full adder. Since the full adder turned out to be too large to make one for each pointer, the two pointers on each datapath share an adder. The arrangement of the two pointer registers and the two increment registers with the carry-select adder is shown in Figure 4.9. The choice of which pair to add is made by 2:1 muxes on the inputs to the adder and inputs to the pointer registers. The control fields for these registers are listed in Appendix A.

The pointer registers are the same as the incrementable registers described above; they are general purpose registers with an extra input to load the adder result. The increment registers are the same as the general purpose registers but without pulldown transistors for the **A** and **B** busses, since they drive only to the carry-select adder. The adder used to increment the pointers is derived from the carry-select adder used in the ALUs.

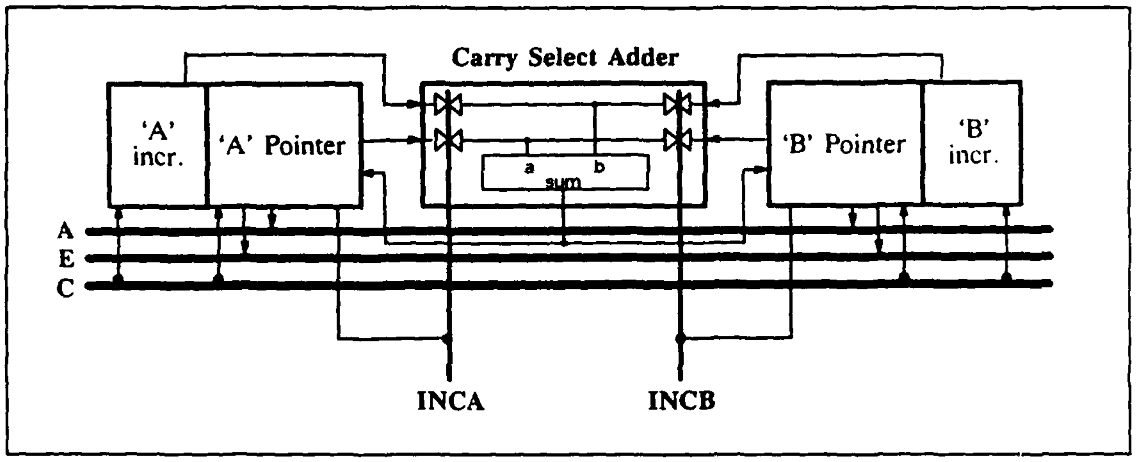


Figure 4.9. Pointer Registers and Adder.

The increment registers are 32 bits wide, so they can hold data if the pointers are being used to manipulate data rather than addresses. The increment registers are isolated from the busses except for being able to load from the C bus. Driving the increment registers onto the data busses would have required a selection choice out of the **A** or **B** register select fields, decreasing the number of selections available for general purpose registers.

4.5.2.1 **Microcode Example.** The microcode shown in Figure 4.10 shows how the two types of incrementable registers support looping, and addressing into a matrix. The routine simply puts whatever value is in the MBRs into every element of the vector whose starting address is in the A pointer (APT). The negative value of the number of elements in the vector is passed in the second upper incrementable register (UIN2). The distance between the elements in the external memory is in the increment holding register of the A pointer.

The first line loads both MARs from the A pointer using the **E** bus tie. At the same time the A pointer is driving the **E** bus, it is also loading its incremented value. The loop counter is checked to see if it has been incremented to zero. If it has, then the calling routine is returned to. With the latency in the control section, the line following the return would be done anyway, which would write the MBRs out to the address in the MARs. If the loop were not done, then the condition on the next line would be true, and the program would branch to the top of the loop.

Again, the latency means that the third line would be executed also. So the loop counter would be incremented towards zero. At the top of the loop, the address of the next element is be

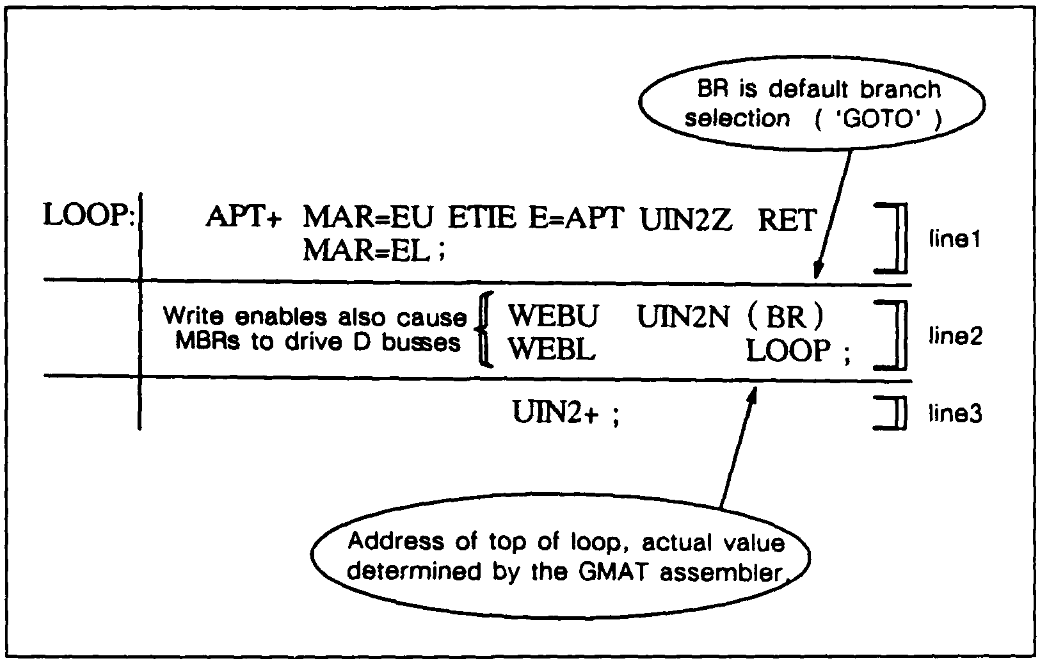

BR is default branch selection ('GOTO')
LOOP:
APT+ MAR=EU ETIE E=APT UIN2Z RET
MAR=EL ;
line1
Write enables also cause MBRs to drive D busses
WEBU UIN2N (BR)
WEBL LOOP ;
line2
UIN2+ ;
line3
Address of top of loop, actual value determined by the GMAT assembler

Figure 4.10. Example of Incrementable Register Usage.

put into the MARs, the pointer bumped up to the following element, and the loop would iterate again.

When the loop counter reaches zero, the second line is still done due to the latency. In this case, the condition on the second line would be false. This is important, because if it were true, the loop address would be loaded into the CAR instead of the incremented return address, and the program would branch back here. The default in every case where the condition is false is the present address plus one. So even though the second line of the loop is being performed, the "present address" further down the control pipeline is the return address of the calling routine, which was popped off the stack by the **RET** instruction. Now the 'next address' due to the false condition is, in this case, the next address after the return address, so the program flows properly.

Notice that the loop counter had to be incremented at least one line before it could be checked. This is because the flags all get delayed one clock cycle before arriving at the condition mux in the microsequencer. This is because the value in the register is not checked until the beginning of the next clock cycle, when it has reached the output of the slave half of the register.

4.5.3 **Instruction Registers.** In order to support the assembly language, the FPASP requires an instruction register to hold the new control bits. Since the instructions must come into the FPASP through R1,R2 or the MBR, it made sense to modify one of these registers to act as an instruction register.

The upper and lower R1's were chosen for this modification since they are next to the open column needed for the data bus. This allows the bus carrying the instruction bits from the R1s to the control section to use the space beneath the data bus. In the case of the upper R1, there are already eight flag lines running out of the register that can double as instruction lines. This design made it easy to fit the assembly language support hardware into the existing design.

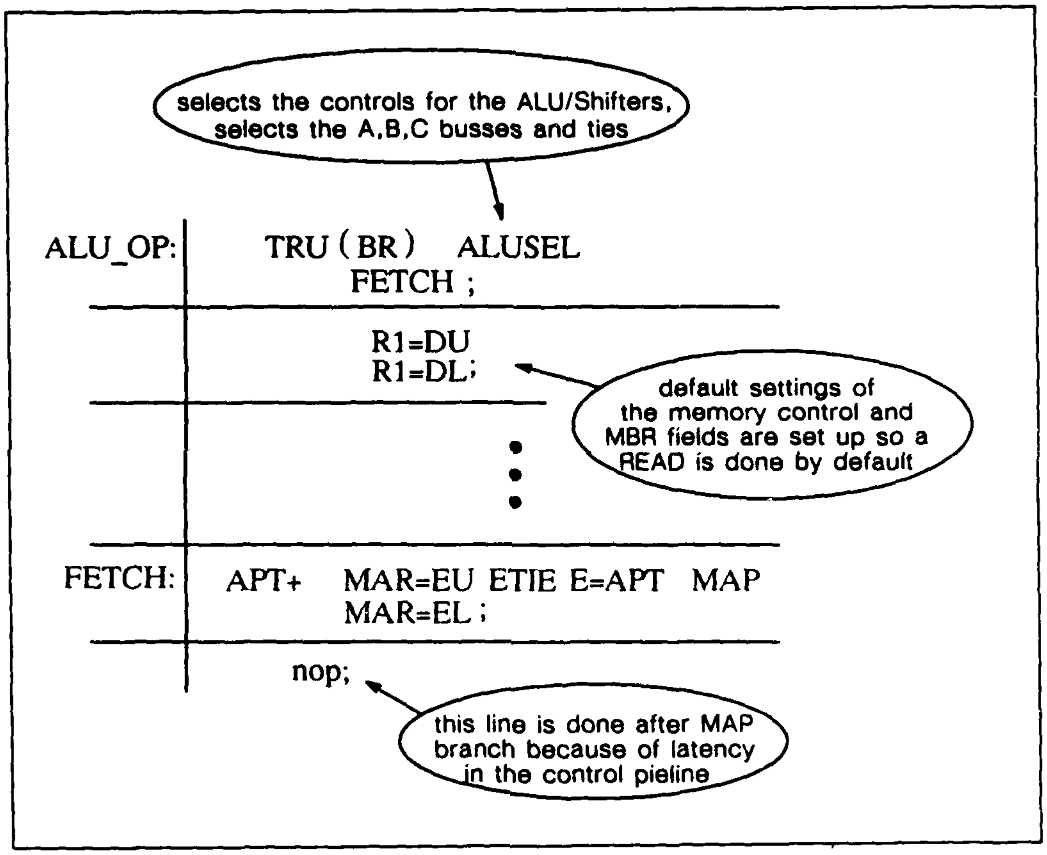
The control bits go from the R1's to muxes on the pipeline registers, where they are chosen over the bits coming out of the microcode ROMs if needed. A special 3 bit field in the microcode word allows eight different sets of muxes to be selected, depending on the instruction being performed. The commands in this field are only used by microcode which supports the assembly language.

4.5.4 **Assembly Language Example.** This section shows the microcode routine used to support one of the assembly language instructions. This is the instruction which does ALU functions. The format for the instruction is shown at the top of Figure 4.11. The upper and lower R1s hold the instruction, and the A pointer holds the address of the next instruction. The microcode that supports this instruction is very simple. When called, it uses the ALUSEL choice in the 3 bit field which controls the muxes on the pipeline. It overrides the control bits from the ROM with the control bits in the R1 registers. Then it loads the next instruction into the R1 registers and unconditionally branches to the FETCH routine.

The FETCH routine uses the Mapping ROM to select the address of the support routine for that instruction. The A pointer is incremented to the following instruction. The NOP in the second line covers the control pipeline latency.

The ALUSEL control selection enables the muxes which override the bus select fields, the bus tie fields, and the ALU/Shifter fields. These fields are fully replac‿d by the bits from the R1 registers, giving the assembly language instruction full choice of all the possible operations that can be done in a single microcode cycle.

The nop is needed at the end of the routine to take up the slack caused by the pipeline latency: the next microinstruction will be done regardless of the branch in the first instruction. The FETCH routine then reads the next instruction into the R1s, and increments the A pointer to the next instruction.



```
ALU_OP:  | TRU ( BR )   ALUSEL
         |        FETCH ;
---------+------------------------------
         |          R1=DU
         |          R1=DL;
---------+------------------------------
         |                 •
         |                 •
         |                 •
---------+------------------------------
FETCH:   | APT+   MAR=EU ETIE E=APT   MAP
         |        MAR=EL ;
---------+------------------------------
         |     nop;
```

selects the controls for the ALU/Shifters, selects the A,B,C busses and ties
default settings of the memory control and MBR fields are set up so a READ is done by default
this line is done after MAP branch because of latency in the control pieline

Figure 4.11. Example of Assembly Language Support Code.

4.6 Processing Components

This section shows block diagrams of the various macrocells which perform the data processing operations on the FPASP. The floating point adder has yet to be fully designed, but the type of cells it will use is known. For all of the macrocells, the decoding logic was simplified as much as possible by arranging the operations in an order that would decrease the number of boolean terms in the control equation. The control options are listed in the microcode definition in Appendix A. Some of these lists will be presented in more detail below.

4.6.1 ALU/Shifter.

The ALUs perform simple arithmetic or logic functions on each of the 32 bit integer data paths. Incorporated into each ALU cell is a linear shifter which can perform a left, right or no shift on the data, with a variety of sources for the bit to be shifted in. The ALU and shifter produce five flags for conditional branching. These flags are held in master-slave flip-flops so they can be used until reset.

The diagram of Figure 4.12 shows the ALU/Shifter and the flags generated by each. Some of the control signals needed for this cell are also shown. These control signals are all generated from the four control bits supplied to each of the two sections.

4.6.1.1 ALU Design.

The ALU for the FPASP is a highly modified version of the one used in the ASP. The new circuitry designed for the FPASP is an input mux to allow selection of A or Abar, and the linear shifter hardware. Also, the "pass" selection was removed in favor of an OR with zero to simplify the decoding hardware. The pulldown transistor for the distributed NOR gate which produces the zero flag was also incorporated in the ALUs.

The operations which can be performed by the FPASP ALU are listed in Table 4.1, along with the boolean or arithmetic function chosen to implement each one. The negative logic functions were implemented with the existing ASP logic gates by using the negative inputs and invoking DeMorgan's theorem:

$$\overline{AB} = \overline{A} + \overline{B}$$
$$\overline{A+B} = \overline{A} \cdot \overline{B}$$

Thus, the negative logic functions have been added without adding more gates, just the Abar choice on the input.

The arithmetic functions are performed in 2's complement fashion using a carry-select adder. This type of adder circuit was required for speed. The tradeoff is that it uses twice as much hardware as a ripple-carry adder.

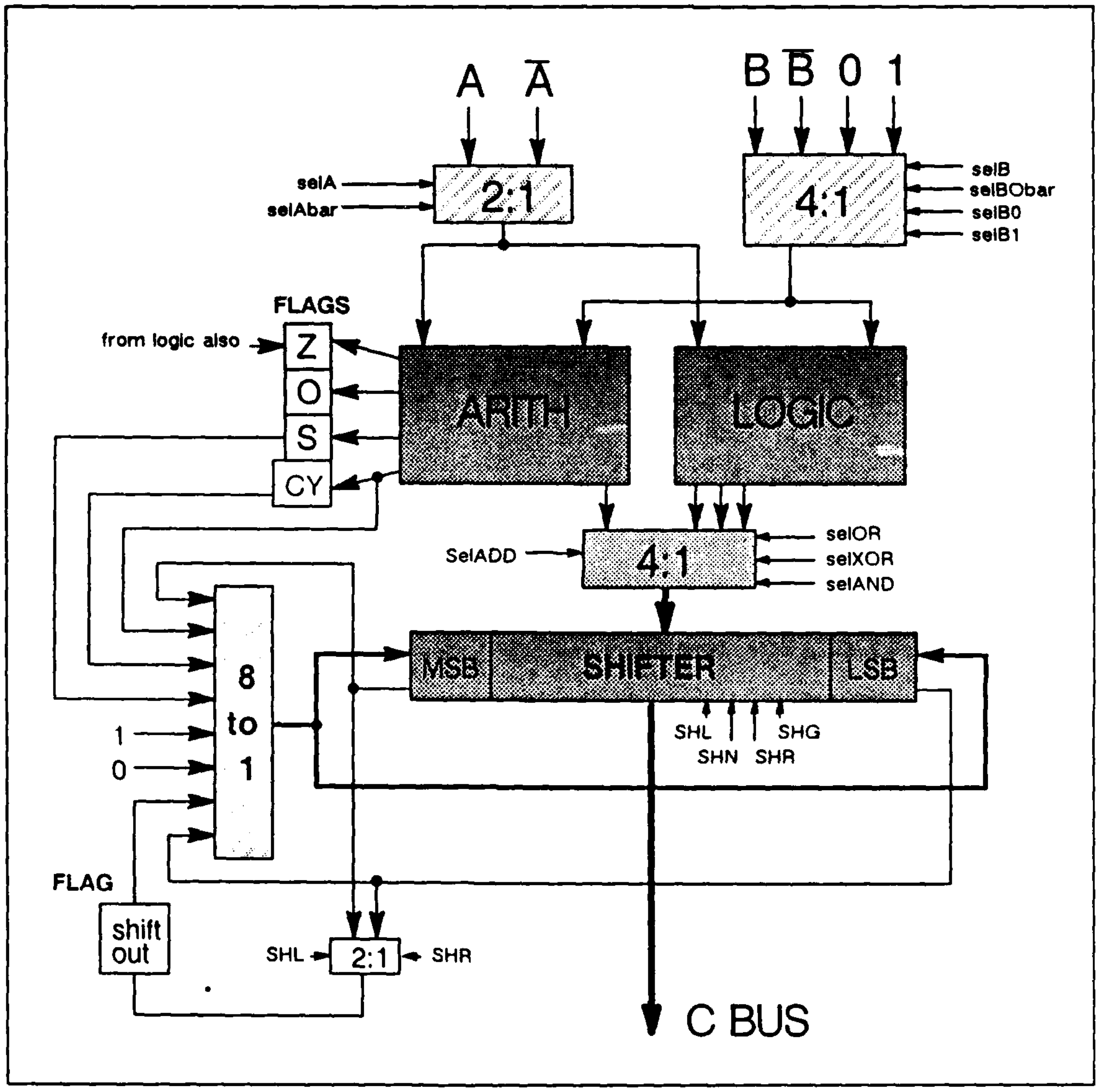


Figure 4.12. ALU/Shifter Block Diagram.

Function	Operation	Flags affected CARRY, OVERFLOW, SIGN, ZERO
MOVN	A	none
OR	$A + B$	zero
AND	$A \cdot B$	zero
XOR	$A \oplus B$	zero
MOV	A	zero
NAND	$\overline{A} + \overline{B}$	zero
NOR	$\overline{A} \cdot \overline{B}$	zero
NOT	$\overline{A}$	zero
INC	$A + 0 + 1$	All Four
SET	$A + B + 1$	All Four, Sets CARRY
ADC	A + B + previous carry	All Four
ADD	$A + B + 0$	All Four
NEGA	$\overline{A} + 0 + 1$	All Four
SUB	$A + \overline{B} + 1$	All Four
SWB	A + $\overline{B}$ + previous borrow	All Four
DEC	$A + 1 + 0$	All Four

Table 4.1. ALU Operations.

A carry-select adder works by computing the sum for both possible carry ins. When the actual carry in becomes known, the proper sums from that stage are selected. The previous carry also selects the corresponding proper carry out of that stage, which is then used to select the sums of the next stage in the adder. The stages used for these carry select muxes are shown in Figure 4.13.

The number of ripple delays can increase in the lower stages because the proper carry takes longer to get to them. The only carry ripple delay which is on the critical path is the one in the first stage, which is only four adders deep. For this first stage, the carry in of the "previous" stage is chosen by the operation being performed.

There are four choices for this carry in bit: the carry of the previous adder operation, its inverse, a '1', or a '0'. The choice required for each operation is shown in Table 4.1

4.6.1.2 **Shifter Design.** The ALU output feeds directly into the Shifter. The Shifter can perform left or right shifts with eight choices of input sources. It can also just pass the ALU output to the C bus, or ground all of the C bus lines. These functions are listed in Table 4.2, along with the shifted in bit needed for each one. The bit shifted out is available as a flag on the next clock cycle.

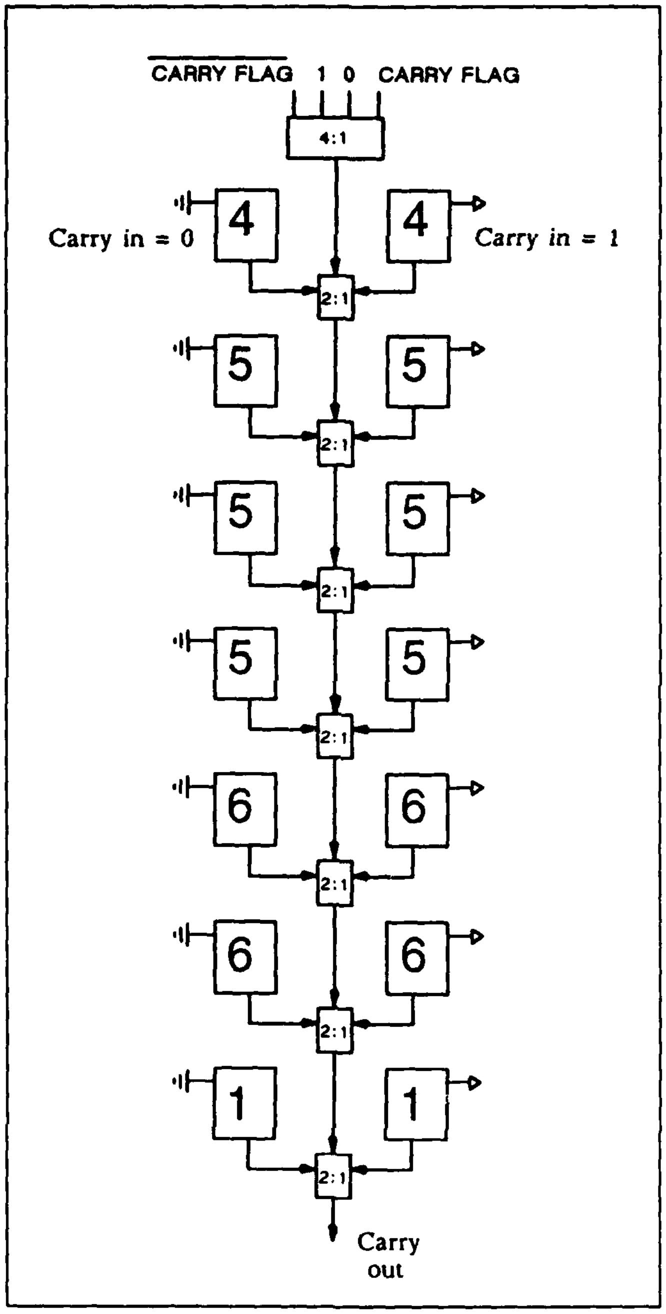


Figure 4.13. ALU Carry-Select Adder.

Function	Shift Direction	Bit Shifted In	Shift-out saved
NOP	No Drive	none (LSB)	no
GNDC	Ground C bus	none (carry flag)	no
PASS	No shift	none (sign flag)	no
SLOT	Left	shift-out flag	yes
SLMS	Left	MSB	yes
SLCY	Left	present Carry	yes
SL0	Left	0	yes
SL1	Left	1	yes
SRLS	Right	LSB	yes
SRCF	Right	Carry flag	yes
SRS	Right	Sign flag	yes
SROT	Right	shift-out flag	yes
SRSE	Right	MSB	yes
SRCY	Right	present Carry	yes
SR0	Right	0	yes
SR1	Right	1	yes

Table 4.2. Shifter Operations.

An 8:1 mux is used to select the shifted in bit. The mux output goes to both ends of the shifter, but there are fewer choices that are useful for shifting in from the left (into the LSB) than from the right. For example, one of the choices available is the most significant bit or MSB. This bit is useful for left circular shifts or for a shift right with a sign extension. On the other hand, the least significant bit (LSB) is useful only for right circular shifts; there is not much need for a "left shift with LSB extension."

The choices of shifts are based on the ones in Mano text. These turned out to be sufficient for the microcode written so far. The only thing the EE588 class needed in addition to what was provided originally in the ALU/Shifter was the ability to see the shifted out bit, which was done by sending it to the condition mux in the control section.

4.6.1.3 **Flag Sources.** Tables 4.1 and 4.2 show which functions set the various flags. These flags are latched into master/slave flip-flops which drive the flag to the rest of the circuitry on the next clock cycle. This latch allows the flags to be held until they are reset, but it introduces a one clock cycle delay from when the flags are set to when they can be checked. The only exception is the carry into the shifter, which has an input for the carry generated by the present function, and an input for the carry saved in the flag register.

The one cycle delay in the flag bit is also required because there is not enough time in one clock cycle for a flag bit from the ALU to be generated and ripple through the condition mux in time to be valid for branching on in the next clock cycle. The effects of the one cycle delay on the microcode were seen the microcode example for the incrementable registers.

The four flags from the ALU are the ones used in the ASP. The first flag is the zero flag, which is raised when the value out of ALU/Shifter is zero. This condition is checked before the value is driven onto the C bus, so the flag can be set without affecting the C bus. The Carry flag is the carry bit out of the last stage of the carry-select adder. If one of the three subtraction operations is being done, the bit coming out of the adder is inverted, becoming a borrow out. The carry out goes directly to the shift-in mux, so it can be used on this clock cycle. It also goes to a master-slave flip-flop to be the carry flag.

The sign flag is just the MSB of the number out of the ALU/Shifter. The overflow flag is the XOR of the carry out of the adder and the carry into the MSB's adder cell. This is the most costly flag to generate in terms of added hardware. To get the true carry into the MSB of the adder requires that a carry- select mux cell be put there, and at the end of the adder, to get the true carry out. This is why the adder ends in two mux cells rather than just one.

The shifted out bit flag is the LSB or MSB of the shifter, depending on whether a left or right shift is being performed. No new control signals had to be generated to load this flag; the shift-right and shift-left can be used directly. This means that a pass or grounding the C bus will not affect the shift out flag.

4.6.1.4 Simplification of Decoders.

The design of a macrocell's control decoder is greatly influenced by how the possible functions it can perform are arranged with respect to the microcode control bits. By aligning the necessary control inputs into groups with similar control bit states, the final boolean equation for those control signals can be reduced. This arrangement of functions into common blocks corresponds to grouping them into contiguous spaces on a Karnaugh map.

The arrangement of choices made for the ALU control fields to simplify the decoding can be seen in table 4.1. With the added ability to choose A or Abar comes the ability to produce the negative logic functions NAND, NOR and NOT, which were not available in the ASP.

This not only adds flexibility to the FPASP ALU, but also allows some choices for the way these functions are carried out. For example, the NOT function can be done five different ways:

$\overline{A} = \overline{A}$	move
$\overline{A} = \overline{A}+0$	OR
$\overline{A} = \overline{A}\cdot 1$	AND
$\overline{A} = \overline{A}\oplus 0$	exlusive OR
$\overline{A} = A\oplus 1$	exlusive OR

The last method was chosen, because after all the other functions were grouped for easy decoding, the NOT function was left in the group of control functions which used XOR and an input of A. Table 4.3 also shows how each logic function in the ALU is used at least twice. This arrangement simplified the decoding of every control bit. The only control bit which could not be fit into symmetric groups was the A input select.

Control Bits	ALU Operation	Function				Inputs Selected		
		+	•	⊕	+	A	B	Carry In
0000	MOVN	x				A	0	don't care
0001	OR	x				A	B	don't care
0010	AND		x			A	B	don't care
0011	XOR			x		A	B	don't care
0100	MOV	x				A	0	don't care
0101	NAND	x				$\overline{A}$	$\overline{B}$	don't care
0101	NOR		x			$\overline{A}$	$\overline{B}$	don't care
0111	NOT			x		A	1	don't care
1000	INC				x	A	0	1
1001	SET				x	A	B	1
1010	ADC				x	A	B	Carry Flag
1011	ADD				x	A	B	0
1100	NEGA				x	$\overline{A}$	0	1
1101	SUB				x	A	$\overline{B}$	1
1110	SWB				x	A	$\overline{B}$	$\overline{\text{Carry Flag}}$
1111	DEC				x	A	1	0

Table 4.3. ALU Decoding Patterns.

Table 4.2 showed the mux selections for all of the operations. The ones which don't need a shift-in bit still cause a mux selection, but it is not used. These selections are shown in parentheses. The same mux selection is made for the upper and lower halves of the table. This eliminates any decode circuitry for the mux selection, the decoding is done by the mux itself, using the three LSB's of the control bits.

The decoding for the type of shift is done by the control MSB, which selects a left or right shift. The only decoders needed are the ones to choose the first three operations, which take replace the left shifts with the less useful shift-in bits. So, although the shifter choice "PASS" is decoded by the shifter input mux to select the sign flag bit, there is no effect on the output, because that choice is also decoded to cause no shift of the bit stream out of the ALU.

4.6.2 **Barrel Shifter.** The barrel shifter was one ASP macrocell that could be used almost directly. Capt. Gallagher's design of the components made it simple to reverse engineer the ASP barrel shifter and use the exact same cells in a larger shifter for the FPASP. Even the PLA decoder could easily be expanded to its full 32 bit width. The barrel shifter performs only left circular shifts, but they are done in a single clock cycle.

Only the C bus drivers were enlarged a little to drive the longer and more heavily loaded lines in the FPASP. A transistor sizing change was also made to the PLA decoder for the "no drive" option.

4.6.2.1 **Control Sources.** A way to insert control bits from the datapath was needed for one of the microcode projects. This was provided by loading the 5 LSB's of the lower C bus into a set of flip-flops. A mux allows the control bits from the ROM or the control bits stored in the register to be used by the shifter control PLA. This is the same method used to support the assembly language, but on a smaller scale.

4.6.3 **Literal Inserter.** The Literal Inserter was one of the first cells to be designed for the FPASP. This was because it originally was to serve also as the A bus tie. This turned out to be unnecessary, and later impractical, when the new floorplan put the control section on the other side of the chip from the A bus tie.

The Literal Inserter is basically the pulldown transistor from the registers cells with a different input. It takes its input from a 16 bit field in the microcode word. It can put those 16 bits in several places. They can go on the upper or lower 16 bits of either the upper or lower or both A busses.

The 16 bits not inserted can also be controlled. The two choices for the uninserted bits are to either ground them to '0', or to leave them untouched at their precharged '1' level. Both of these choices are useful in different situations. The most common uses of a literal inserter are to put a constant or a mask on the bus. For constants the MSBs will probably be zeroes, while for a mask they will probably be left as ones.

The Literal Inserter has its own control field to drive the **A** busses. This means it can drive the bus at the same time as a register. This usually means a problem in the code, but it can be used also for masking out bits of a register word while it is still on the **A** bus. So the data can be masked and then operated on in the same clock cycle. By driving two signals onto the precharged **A** bus, an AND operation is performed, without any fighting between pullup and pulldown transistors.

The subcells which make up the Literal Inserter are identical. This cell is shown in Figure 4.14; which also shows the simple decoding circuitry which resulted from the arrangement of the operation choices. The design of the cell and the arrangement of the control choices simplified the design of the control decoders down to a few gates.

The cell itself is just a pulldown transistor controlled by a 2:1 mux and an enable line. One of the two mux choices is a bit made up from the control bits, which says whether to ground the bus or not. The other is the bit out of the microcode word to be inserted on the bus.

4.6.3.1 **Saving the Flags.** The literal inserter is used by the routine which saves the state of the machine when an internal inconsistency forces the machine to halt. This can either be a call from the user's code, or a forced branch from an internally generated signal. The TRAP routine uses the literal inserter to write out the flags.

This was made possible by putting the literal inserter by the control section. The literal inserter provides an easy path to get bits out of the control section and onto the datapaths, where they can be written out to memory. The number of literal bits is also conveniently the number of inputs to each bank of the condition mux, which is described later.

The three fields of the literal inserter that are not used for regular operations are decoded to mux the three banks of flags in place of the literal field. In this case, the flags can only be sent to the upper A bus, where they appear either on the upper or lower 16 bits. The 16 unused bit are zeroed out or left alone according to the literal inserter controls. It makes no difference in this case what the other bits are, just that the flags are saved off the machine. Of course, any function in the microcode is available to the user, so the flags could be saved at any time if required. This would be useful for self-test code.

4.6.4 **Function ROM.** The function ROM is a small version of the LPROM used for the microcode. This was the simplest way to make the choice of seeds user-programmable. The two functions which will be written into the XROM, square root and invert, will have fixed seeds. These values can be "pre-zapped" by putting the diffusion in place before fabrication. This, way the seeds will exist without having to be put there by the laser.

The block diagram of the Function ROM is shown in Figure 4.15. Since the ROM only puts out five bits, the PLA cells and bitline drivers can be made much smaller than the ones needed for the microcode store.

The Function ROM takes in three control bits and five bits from the datapath. The ROM puts out the four MSBs of the seed's mantissa and the LSB of the seed's exponent. The three control bits choose which set of seeds will be used, and the datapath bits select the particular seed.

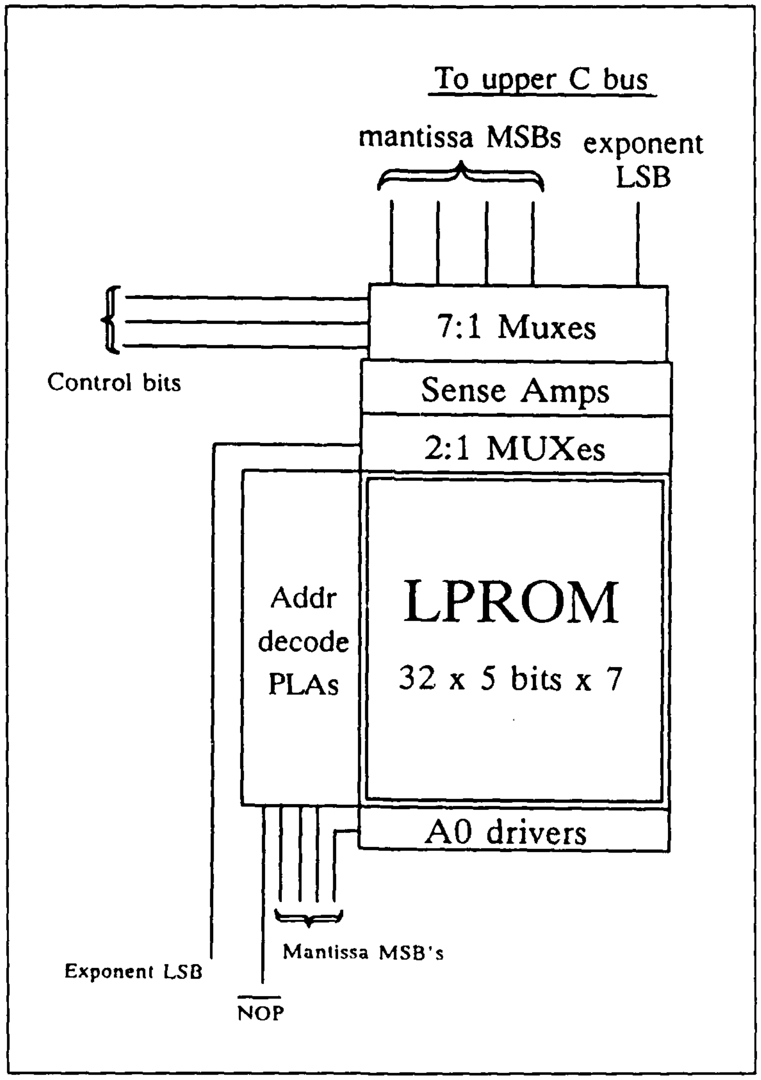


Figure 4.15. Function ROM Block Diagram.

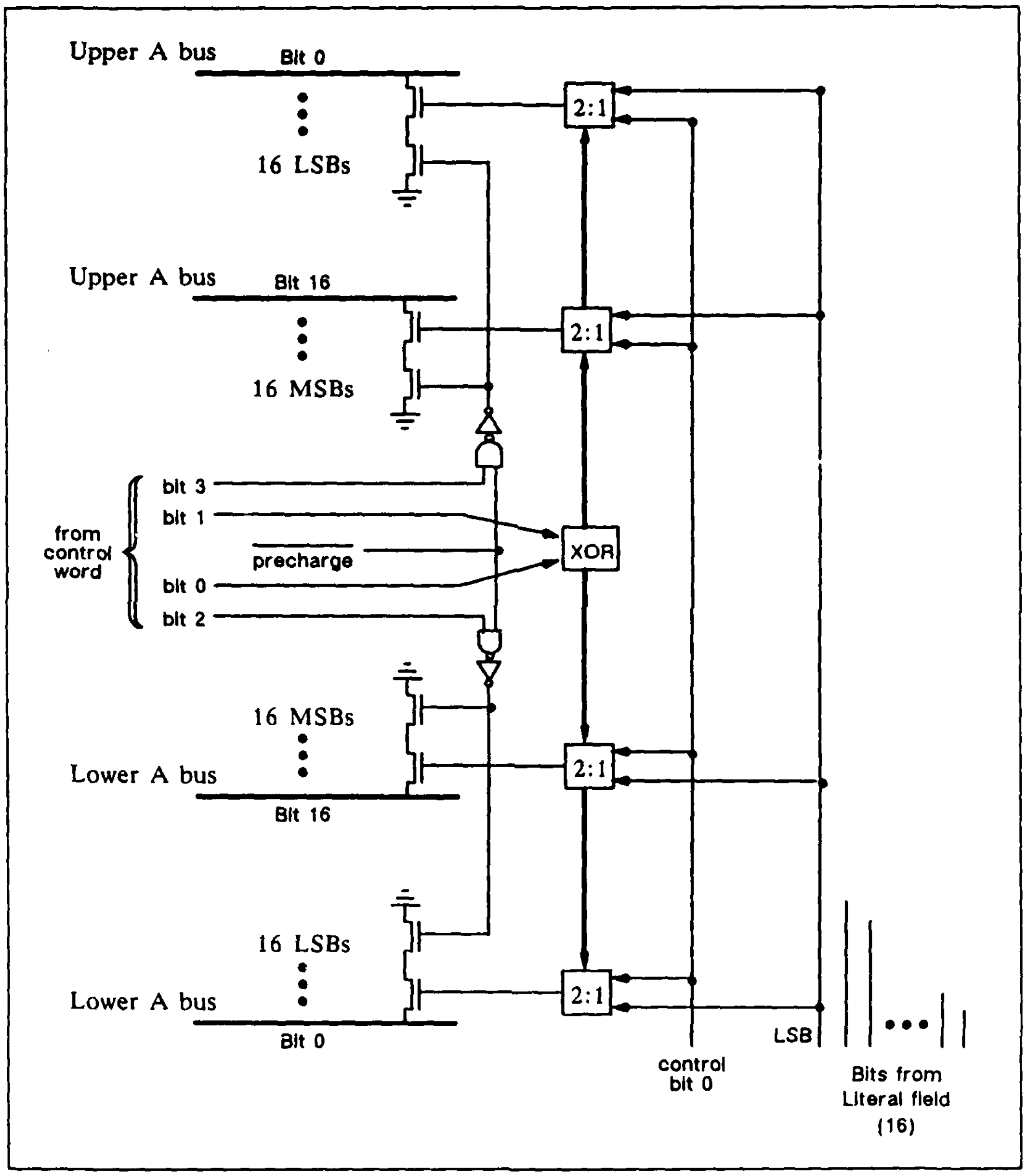


Figure 4.14. Literal Inserter Circuitry

The number of tables can be increased for functions which do not require the seed's exponent bit. This can be done by having the routine set the 'exponent' input bit, and then using only the four mantissa MSBs to address into the table. Similarly, the number of seeds for a function could be increased by taking the five MSBs of the mantissa and shifting them left one bit.

The upper **B** bus provides the input bits, and the result is driven onto the upper **C** bus. Only the five lines corresponding to the mantissa MSBs and exponent LSB are driven. The rest of the bits must be set by another source, usually the literal inserter.

Microcode written for the Newton-Raphson inversion routine is listed in Appendix B. This code shows how the Function ROM is used to generate the seed MSBs, and how the literal inserter is used to create the final floating point seed.

4.6.5 Floating Point Hardware.

4.6.5.1 **Multiplier.** The multiplier uses Booth's octal encoding and a central core of adders to do the mantissa multiplication [Fre88]. The **A** bus input is multiplied by three, then that result and the original number are fed into the array. The **B** bus input is octally encoded according to Booth's algorithm. The **A** number times -4,-3,-2,-1,0,1,2,3 or 4 must be added to the partial product depending on the encoding of the **B** number. Once the multiplying factor is known, one of the two **A** inputs can either be added directly, shifted once and added, or the 2's complement of the number could be shifted and added.

As the partial product reaches the bottom of the array, a 110 bit tree carry-select adder is used to form the final product. The final product is rounded off and combined with the exponent, which was calculated in a separate circuit. That circuit forms the exponent by adding the two input exponents, taking into account the bias of 1024. The equation of the final exponent is EXP A + EXP B - 1023.

The core adder array can also be configured to multiply 32 bit integers, giving a full 64 bit result. The particular operation to be performed is signalled to the multiplier when the data is loaded into the input registers.

Flags required by the IEEE standard are also generated and output either as distinct flags to the condition mux, or encoded into the result. The overflow flag is also used to indicate if an integer multiply result has MSBs on the upper datapath.

4.6.5.2 **Adder/Subtractor.** The floating point adder/subtractor needs only a single carry-select adder like the one in the ALU to form the mantissa of the result. The exponents require more manipulation than in the multiplier, though. In the case of the adder, the exponents must be made equal before the mantissas can be added together.

The difference between the exponents is used to tell a barrel shifter how far to rotate one of the mantissas before they are added together. Once the exponents are aligned, the exponent of the result is known. If the difference between the exponents is so great that the smaller mantissa is barrel- shifted completely away, a flag is raised.

4.6.5.3 **Interface and Timing.** The interface to the rest of the FPASP is identical for both of the floating point processors. The input data is loaded into a master-slave flip-flop, and computation begins at the start of the following clock cycle. At the end of the clock cycle following that, the circuitry has settled and can be enabled to drive either the **B** or **C** busses. Control of bus access is by the same decoders used for the registers. The multiplier and adder are treated just like registers when it comes to output. Figure 4.16 shows the interface hardware common to both floating point processors.

The main difference between these circuits and the registers is that the **B** bus is driven instead of merely being pulled down or left charged. This allows the full clock cycle to be used, otherwise the circuit would have to settle before the end of precharge on the cycle that the drive control for the **B** bus is issued. If the **B** bus were not driven it might be discharged accidentally while the circuitry settled in the time after the fall of precharge.

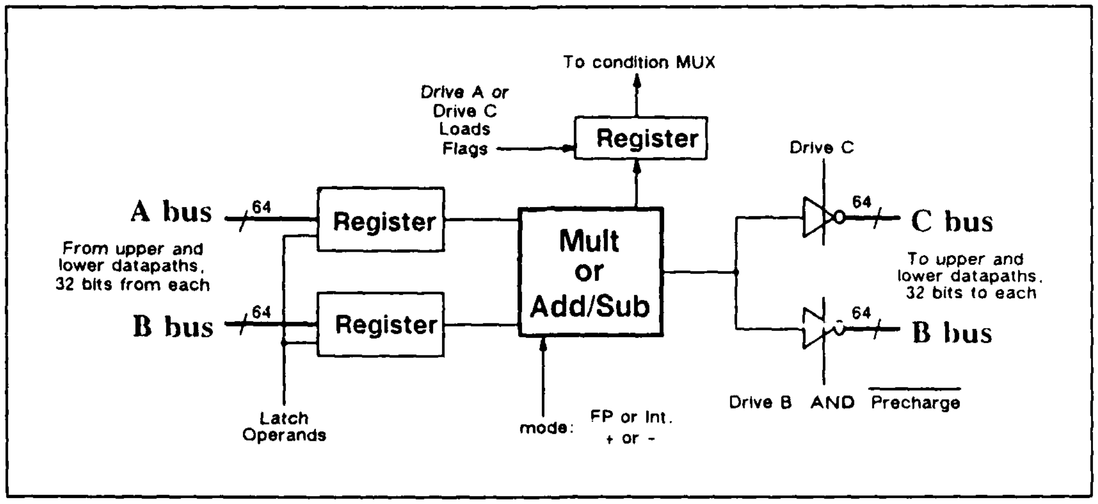


Figure 4.16. Floating Point Hardware Interface.

Figure 4.17 shows the operating cycle for both the multiplier and adder. Once they have been loaded the first time, they can be loaded and have their previous results driven out at the same time every two clock cycles. If the input is loaded, the operation will be started on the next clock cycle and the results will begin changing. If the inputs are not loaded, the results can be held as long as needed.

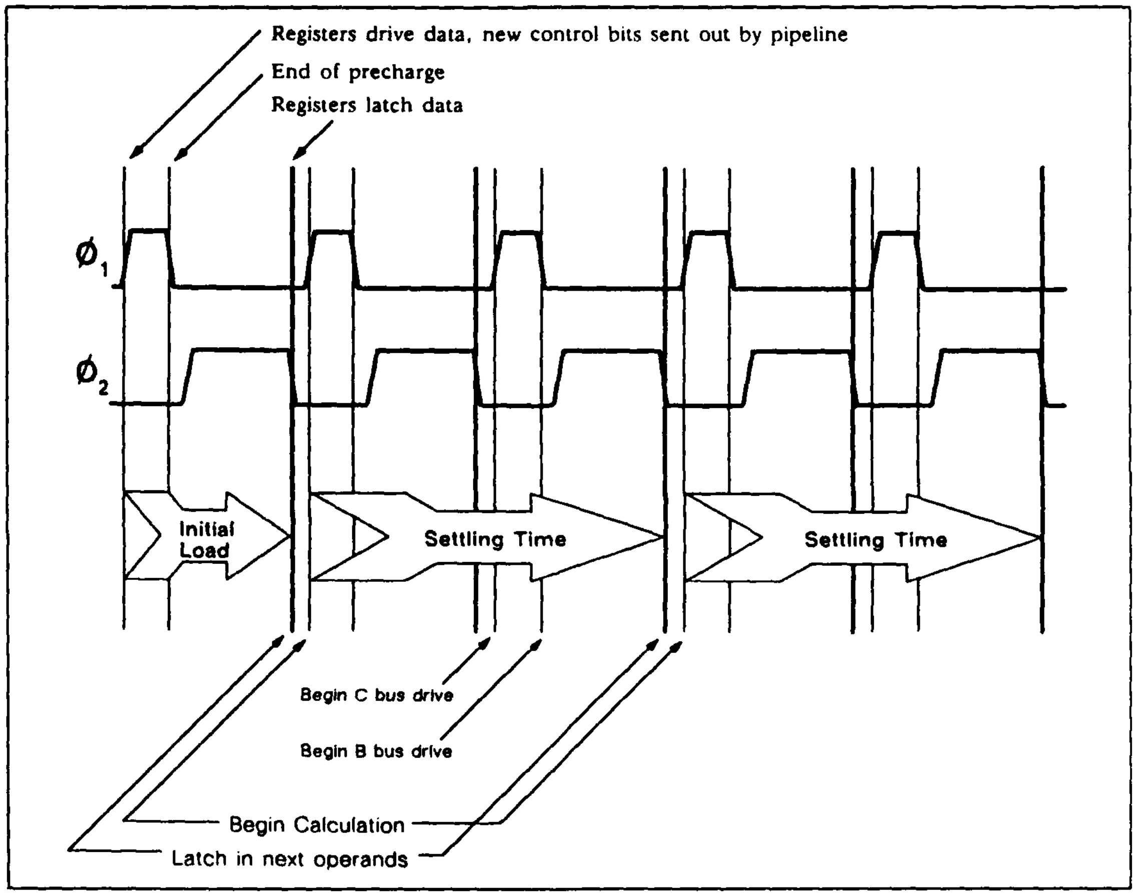


Figure 4.17. Floating Point Hardware Timing.

Each of the floating point processors has a set of flags which can be checked for either the true or false condition. In addition, there is a flag which is a combination of the overflow and not-a-number flags for a single check of any abnormal results. This is the flag used in loops where there are not enough lines of code to check each condition separately. The flags are valid on the clock cycle after the results are driven out. The flags remain unchanged until the next floating point operation. All of the flags available are listed in the microcode definition in Appendix A.

4.7 Microsequencer

A detailed block diagram of the microsequencer is shown in Figure 4.18. This is the most important and complex part of the FPASP, and went through the most revisions. The basic sequencer is still the one from the Mano text, but it no longer uses any of the cells from the ASP except the registers in the stack. Major changes include the extension of the stack into external memory, and the mapping ROM needed for the assembly language.

4.7.1 **Branch Control.** The flow of the microcode program is controlled by the choice of addresses into the XROM/LPROM microcode store. This address comes from the Control Address Register (CAR). The address to be loaded into the CAR is selected from four choices, based on the input from the control word, and control signals generated within the FPASP. Each of the choices has its own source and control logic for selection.

The decision of which address to load is made by the branch control logic which drives the 4-to-1 mux at the input to the CAR register. Three of the choices are conditional: branch (BR), return from a subroutine (RET), and call a subroutine (CALL).

BRANCH and CALL take their next address from the microcode word. These bits are the 10 LSBs of the field used for the literal inserter, so that device cannot be used on the same line as one of these two conditional branches. The return takes its address from the stack, where it has just been 'popped' to the top.

The address from either the microcode word or the stack is loaded only if the specified condition is true. If the condition is not true, the default address for all three branches is the incremented value of the CAR. This is the address of the next sequential instruction in the microcode. For this reason, two of the choices into the condition mux are an unconditional true and false.

If a branch must be made, then the unconditional true is chosen. If the code is to be done sequentially, the only way to specify the next address is to use one of the branch conditions and choose the unconditionally false flag. In the FPASP, the default values for these two microcode fields were chosen for the latter case, branch and unconditional false. The program will proceed sequentially if no value is put into these two fields in the microcode.

The MAP selection is not conditional. If this branch is selected, it will occur regardless of any flag specified, including the unconditional true or false. The map choice is used for two purposes: to support the assembly language, and to allow the FPASP to do a hardware call to the TRAP routine.

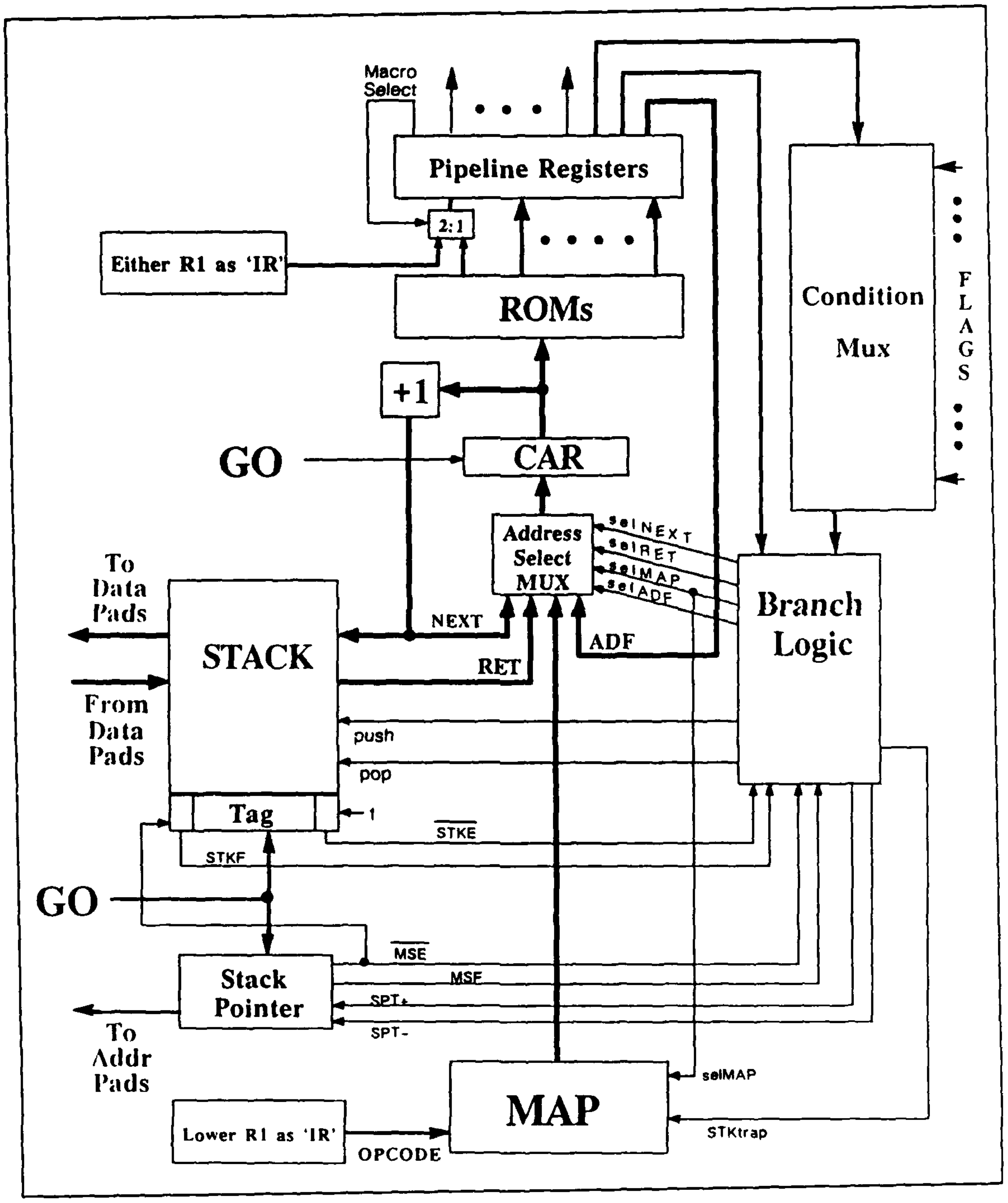


Figure 4.18. Microsequencer Details.

The latter will occur if an inconsistency is found in the stack control signals. The conditions for this to occur can be seen in the control logic equations in Table 4.4. The conditions checked for are whether a call is being made when the external stack is full, or whether a return is being made when the internal stack is empty.

Source of Next Address	Conditions for Selection
Incrementer	$\overline{\text{COND}}\cdot(\text{BR} + \text{RET} + \text{CALL})$
Stack	$\text{COND}\cdot\text{RET}$
ROM field	$\text{COND}\cdot(\text{BR} + \text{CALL})$
Mapping ROM	$\text{MAP} + \text{STKTRAP}$
STKTRAP (goto trap)	$\text{COND}\cdot(\text{CALL}\cdot\text{MSF} + \text{RET}\cdot\text{STKE})$ (these are inconsistent conditions)

Table 4.4. Next Address Sources for CAR.

Other inconsistencies were originally checked, such as whether the internal stack was indicating it was empty when the external stack was not. But there are many of these less likely conditions, and to check them all would delay the control bit too long. So the only conditions checked are those which will arise if the hardware is operating correctly, but the software has exceeded its capabilities.

The box labelled branching logic also provides the control signals for all of the rest of the microsequencer. These include the signals for controlling the stack and stack pointers. The one control not produced by this logic cell is the **GO** signal. This comes from the host through a dedicated pin. It resets the CAR, the pointer to the external stack, and the tag field of the stack to zero.

4.7.2 **Branching Latency.** The latencies caused by the control section registers can best be seen in the selection of the branch condition. This is illustrated in Figure 4.19. The control bits which select the flag are in the instruction loaded two clock cycles ago. This is why the instruction after a true branch condition is still executed: it was in the middle of the pipeline between the instruction which selected the branch and the instruction which the branch selected.

The microcode ROMs are fast enough to produce the condition mux selects and branch selects in time to choose the next address for the CAR without going through the pipeline registers. This would remove the problem of having the instruction after the branch executed. The reason this was not pursued is that the ROMs put out erroneous control bits during the precharge pulse. These bits could cause the stack to be popped or pushed, and the state of stack would then be incorrect.

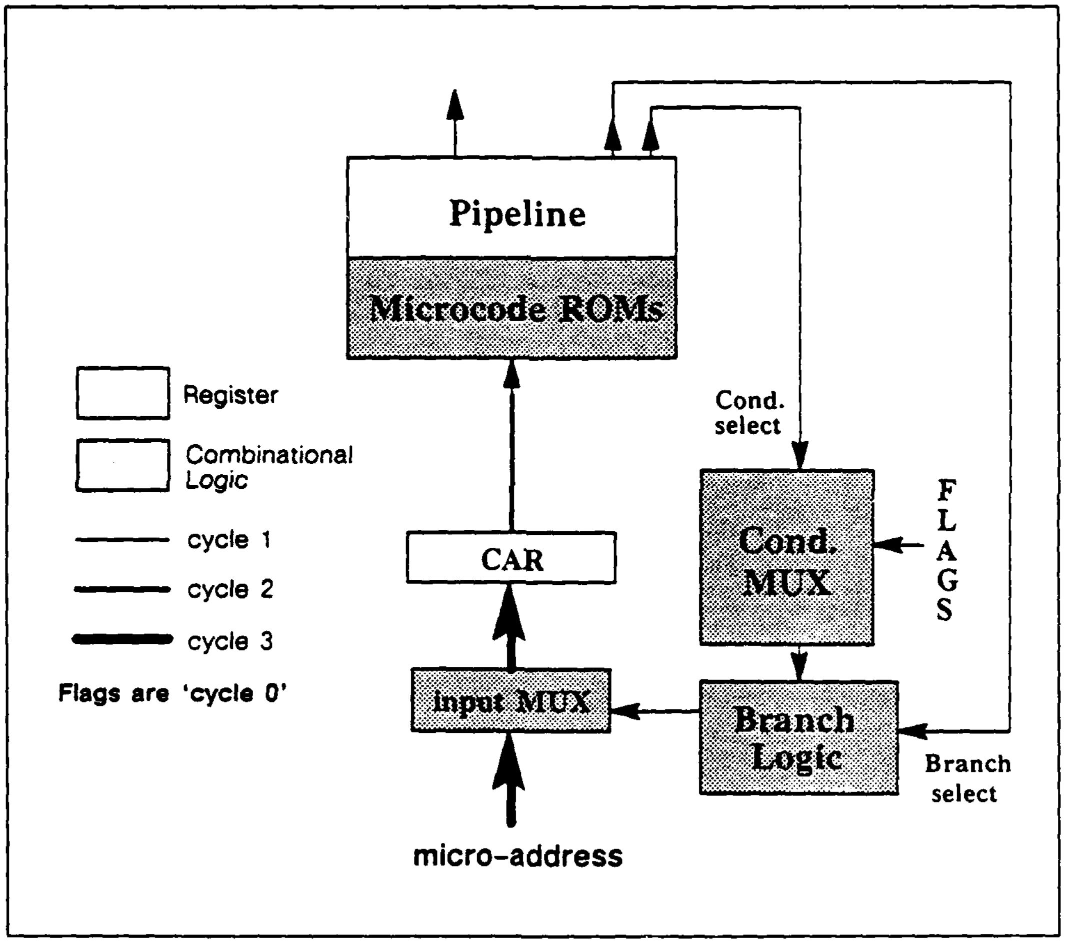


Figure 4.19. Microsequencer Latency

4.7.3 **Condition Mux.** The condition mux collects all of the flags generated throughout the FPASP. The mux is then used to select which flag will be the branch condition. If the selected flag is a '1', then the condition is true. The select lines come directly from the microword on the control bus.

The condition mux is made up of three 16-to-1 muxes as shown in Figure 4.20. These allow a total of 48 distinct flags to be selected from. The 6 bit "Conditional Multiplexer Select" field of the microword allows 64 choices. The 16 extra selections are the inverses of the flags fed into the first of the 16-to-1 muxes.

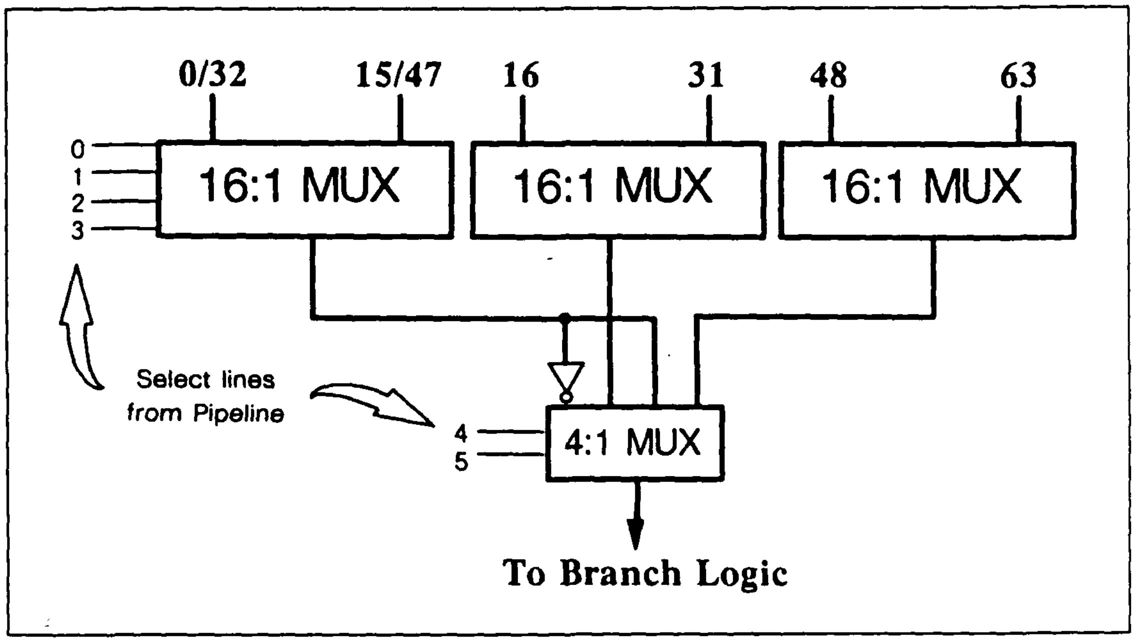


Figure 4.20. Condition Multiplexer.

There are actually more than 16 flags whose inverse can be used as a branch condition. For these extra ones, it was easiest to use two inputs of the other 16-to-1 muxes. Doing it this way allowed a single design for all three muxes, and it simplified the decoding of the control bits. With the 16-to-1 muxes and the 2-to-1 polarity reversing mux arranged as shown, no decoders are needed at all. The control bits from the microword are decoded by the muxes directly.

4.7.3.1 **Flag Sources.** Some of the flags do not come from the datapath. Two of the flags come directly from pins. These are the IO1, IO2 flags. They are provided for testing and can also be used for interrupt flags.

Other flags which do not come from the datapath are the unconditional true and false. These are formed by grounding the first input of the 16-to-1 mux that has the polarity reversal mux after it. The default condition mux control word is 000000, which is the false condition.

Another pair of flags are the 'even' flags. These flags come from the datapaths, but not from the processing hardware. They are merely the inverted LSBs of the C busses. This flag was added at the request of one of the EE588 groups so they could easily check whether the integer on the bus was even or odd. If the number is even, this flag will be 'true.'

These LSBs are passed through master-slave flip-flops to ensure they will be valid in time to affect the branch logic. This also means they are not available until one clock cycle after they generated, just like all the other flags. The reason for this is that the worst case delay through the ALU/Shifter, combined with all the delays in the branch logic, are longer than the clock cycle, which was made long enough only for the worst case delay of the ALU/Shifter alone.

The even flags must be checked on the cycle after being generated or they will be lost. This is because they are not held in the flip-flops, only delayed by them. The flip- flops latch in the LSBs of the C busses on every clock cycle because no control bits are available to control them. So the flag only remains valid until C bus is driven by another bit.

The IO1, IO2 and the unconditionally true and false flags are the only ones which do not pass through flip- flops before reaching the condition mux. For the IO flags, this means they cannot be saved by the TRAP routine unless they are valid when the first group of flags are put out by the literal inserter.

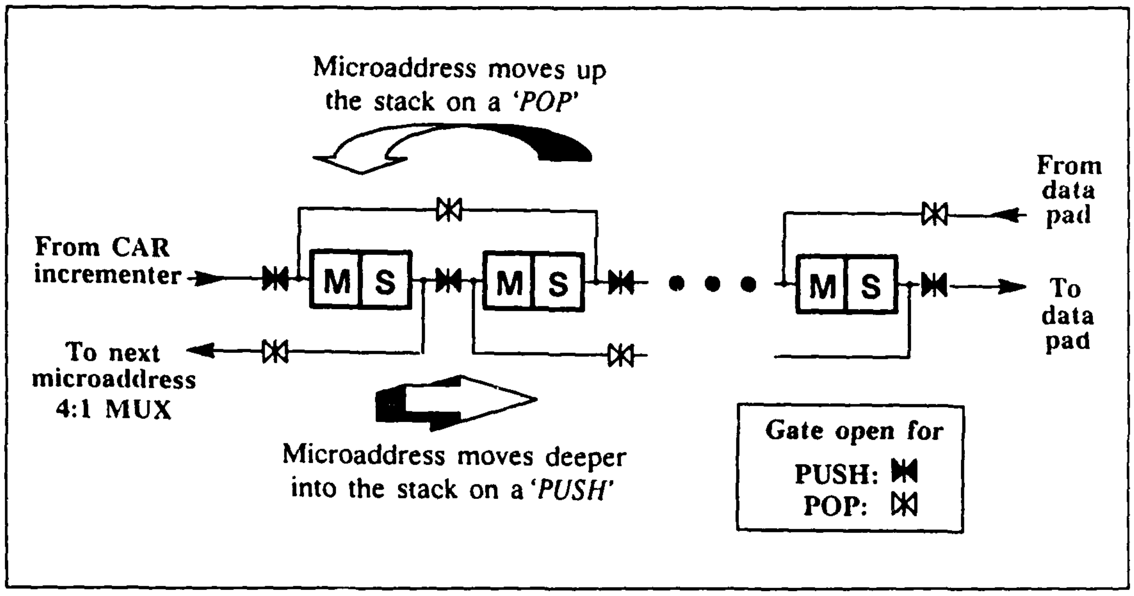
4.7.4 **Mapping ROM.** The mapping ROM (MAP) uses the opcode of the assembly language instruction in the lower R1 to select the micro-ROM address of the routine which supports that instruction. The MAP is an LPROM similar to the one used for the Function ROM. This allows the user to map the unused opcodes to the addresses of microcode routines for new assembly instructions.

There are 32 possible micro-addresses in the MAP, corresponding to the 32 combinations of the 5 opcode bits. Ten of these are used for the fixed assembly language instructions. The zero address is used to point to the TRAP routine, so the machine can make a hardware call to save the state of the machine. That address is also accessible from the microcode by clearing the lower R1's 5 LSBs and using the MAP branch instruction.

To do the hardware call the microsequencer branch logic closes the gates from the lower R1 and grounds the control inputs. To keep the LPROM from being addressed on every clock cycle and thus dissipating power, the MAP signal is used as one of the inputs to the PLA cells.

An LPROM was chosen for the MAP since it is the only laser programmable cell available at this time. The MAP is organized as 32 by 10 bits. The fixed addresses in the MAP are fabricated with the diffusion already in place, just like the fixed seeds in the Function ROM.

4.7.5 **Stack.** The stack is an array of master-slave flip-flops 11 columns wide by 16 rows deep. Each flip-flop in a column can load the output of the flip-flop above or below it. This forms a last- in, first-out (LIFO) stack. Extra hardware has been added to extend the stack into the external memory when it has filled up. The arrangement of flip-flops is shown in Figure 4.21.



Microaddress moves up the stack on a 'POP'
From CAR incrementer
M
S
M
S
M
S
From data pad
To data pad
To next microaddress 4:1 MUX
Microaddress moves deeper into the stack on a 'PUSH'
Gate open for
PUSH: ⧓
POP: ⋈

Figure 4.21. Stack Register Connections.

The stack holds the return addresses of subroutines. When a CALL instruction is issued and the condition is true, the address of the subroutine is loaded into the CAR, and the address that was in the CAR previously is incremented and 'pushed' onto the stack.

When the subroutine is done, it issues a RET instruction and chooses a true condition. This causes the CAR mux to pick the address off the top of the stack. At the same time, the stack flip-flops load the output of the flip-flop below them, 'popping' each of the stored addresses up one row.

4.7.5.1 **Extension to Memory.** Extending the stack into the external memory required additional circuitry to keep track of the addresses of the micro-addresses put into the external memory, and to tell when to read or write that data. Figure 4.22 shows the hardware associated with the stack extension. The addresses are produced by an up/down counter called the stack pointer. The signals controlling the read and write come from the branch logic and from the tag column of stack flip-flops.

The tag column is used to keep track of how deep the stack has been pushed. When the GO pin is raised by the host, this column of flip-flops is reset to zero. The input to the top of the tag column is always a **1**.

When an address is pushed onto the stack, the **1** tag is pushed on with it. As the stack fills, the **1**'s eventually reach the last flip-flop in the column. This becomes a signal to the control logic that the next push will cause the stack to overflow. At the bottom of the tag column, a **1** or **0** is popped in, depending on whether or not the external stack is full. A zero is loaded if the stack pointer is zero.

If the tag says the stack is full and another push occurs, the control logic activates a set of muxes to extend the stack into the external memory. The address pushed out of the stack is muxed onto the 10 LSBs of the lower data pads, the address in the stack pointer is muxed onto the lower address pads, and the lower write enable pin is dropped. This writes the stack address into the lower memory at address zero. The stack pointer is then incremented to point to the next address.

If there are stack addresses in the external memory when the stack is popped, the decremented pointer address is muxed onto the address pads, but in this case the write enable pin is left at its default high level, so a read is performed. The data is muxed from the pads directly into the bottom of the stack.

This takeover of the address and data pads by the stack means the microcode cannot be trying to do a read or write on the same line as a call or return. This does not apply to routines which will not cause the stack to overflow, but the user must make sure those routines are not called by others that might cause an overflow.

4.7.5.2 **Stack Pointer.** The stack pointer consists of two registers: one with an incrementer, and the other with a decrementer. It also has two logic gates for determining whether it is full or empty. The hardware is shown in Figure 4.23.

The two registers work in a master-slave fashion. The master is the register with the incrementer. The address which is sent to the pads is determined by whether a pop or push is being

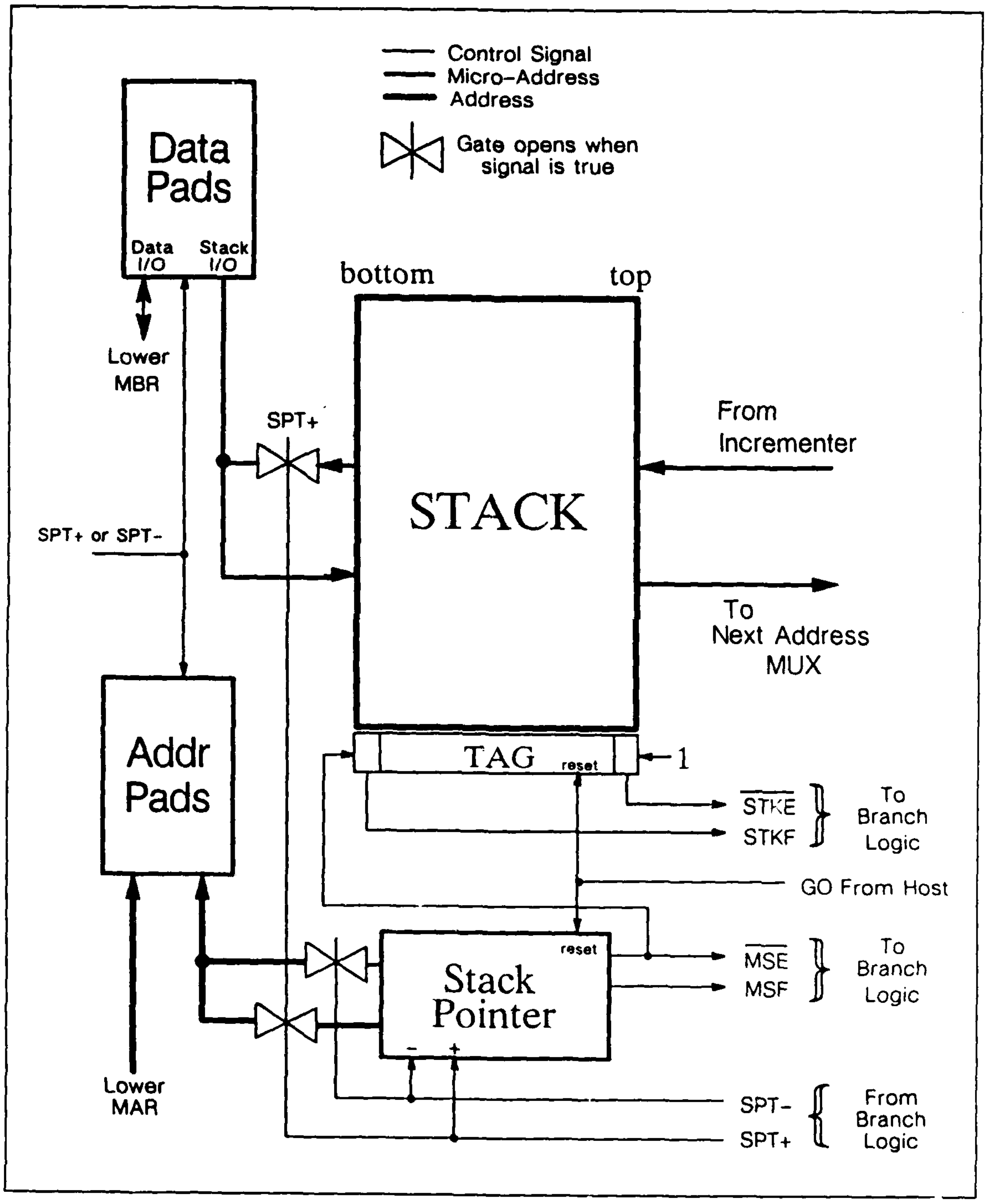
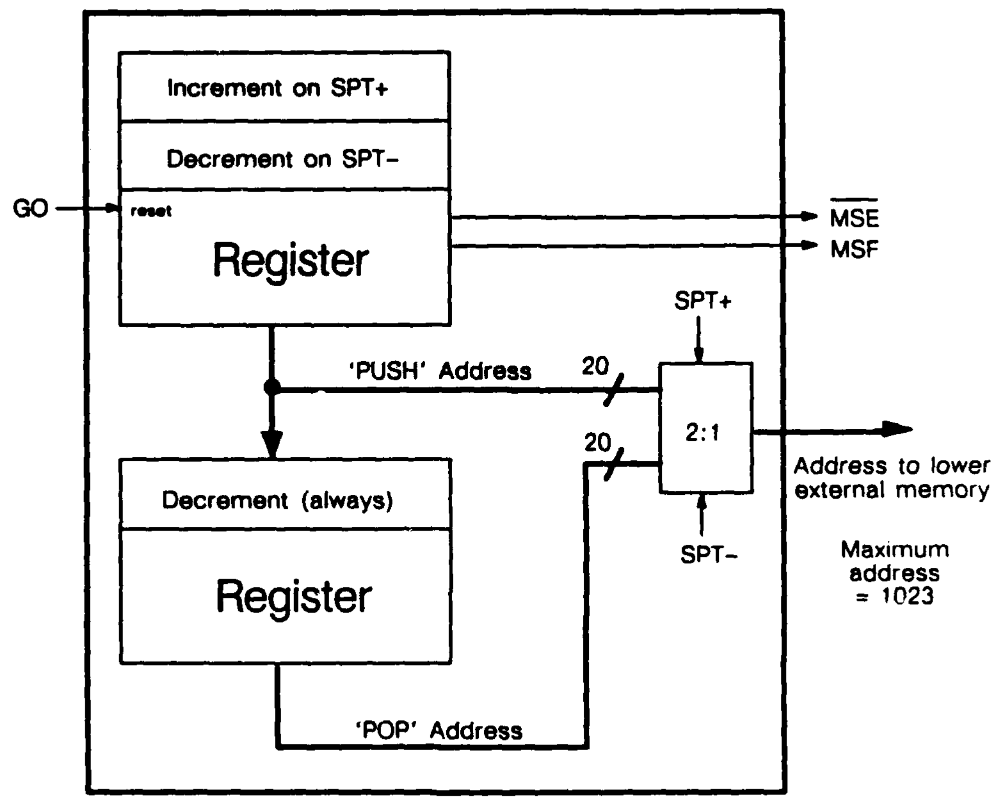


Figure 4.22. Stack Extension to Memory.



$\overline{\text{MSE}}$ = Memory Stack not Empty
set if any register bit = 1

MSF = Memory Stack Full
set if all register bits = 1

SPT+ = Increment stack pointer
(push microaddress into memory)

SPT− – decrement stack pointer
(pop microaddress from memory)

Figure 4.23. Stack Pointer Circuitry.

done. To keep track of the proper addresses, the master register always loads whichever address was sent to the pads. The slave register always loads and decrements whichever address is being put out by the master register.

The slave is a register, so there is a one clock cycle delay from when it loads in the decremented value and when it can drive it out again. This is not a problem because two returns cannot occur one after the other due to the latency in the control pipeline.

Two gates determine if the external stack is full (MSF) or empty (MSE). The MSF flag is used to generate a trap condition if a push is attempted after the external stack is full. The MSE flag is used to feed the tag column. If the external stack is not empty, a 1 is popped into the tag. The bit into which the 1 is popped is itself a flag: stack full (STKF). Thus the internal stack reads full as long as the external stack is not empty. When the external stack is empty, 0's are popped into the tag column. When these 0's reach the top of the stack they indicate that the entire stack is empty.

With a register 10 bits wide, a total of 1023 external memory words can be used for stack values. The stack values go into the 10 LSBs of each memory word, and it does not matter what goes into the other bits. For the addresses, the 10 LSB's are set by the pointer, and the 10 MSBs are set to zero. Only 1023 words can be addresses instead of 1024 because on the last push With this type of pointer, the external stack can be made any depth, as long as the incrementer and decrementers can settle in one clock cycle. The stack pointer is reset to zero by the GO signal from the host.

4.7.5.3 **Reserved Memory Locations.** The stack and the TRAP routine require some reserved space in the external memory. The easiest part of memory for the hardware to address is the lowest part, since it is simple to clear a counter to zero and start counting up. The first word of the upper memory is used to tell the FPASP if it has been programmed when the machine starts up. It is overwritten if the TRAP routine is called.

The stack writes out to the lowest 1023 addresses of the lower external memory. The TRAP routine writes the state of the machine into the lowest addresses of the upper memory.

4.7.6 **Microcode Store.** The microcode is stored in two places on the FPASP. Fixed code is fabricated in the XROM, and user-defined code is written into the LPROMs. These two parts of microcode memory share a contiguous address space, so the microsequencer does not care where the next instruction comes from. The memory is organized as 784 words deep by 128 bits wide. There are actually two halves of the microcode ROM, each one 64 bits wide.

There are 640 words of XROM and 144 words of LPROM. The address lines to the two of them decode which control word to read and which of the two ROMs to read it from. The two types of ROM are completely separate electrically; a mux is used to select which one will output the control word. Figure 4.24 shows the arrangement and interconnections between the ROMs.

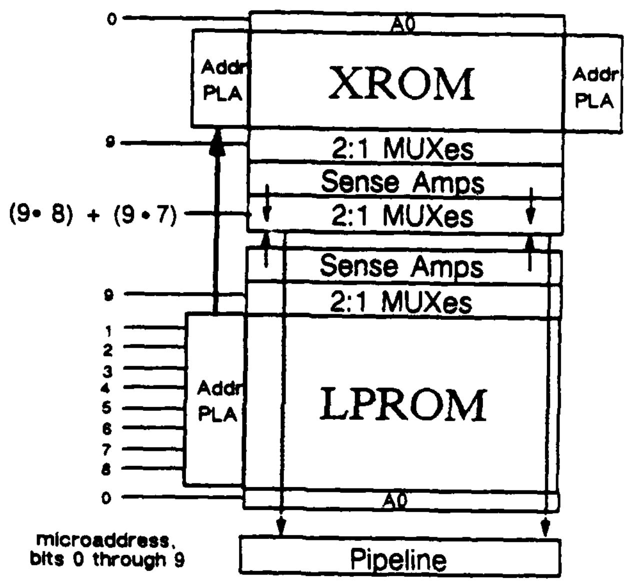


Figure 4.24. Microcode ROM Arrangement.

4.7.6.1 **XROM.** The XROM has been described in other papers, so it will not be covered in great detail here [Ros85] [Lin88-2]. It is a precharged device which uses the presence or absence of a transistor to represent data. The arrangement of four transistors around a common drain gives the cell its name.

Figure 4.25 will be used to show how the XROM works. The addresses are decoded in NAND gates along each side of the array. While the decoding is being done, the bitlines are precharged high. When precharge ends, the wordline associated with the decoded address goes high. This wordline goes to all the gates of the transistors for that word and the word next to it. These pairs of transistors have their source tied to one of two lines: one line is driven by the address LSB and the other by its inverse. The line which is low decides which transistor of the pair is the actual data.

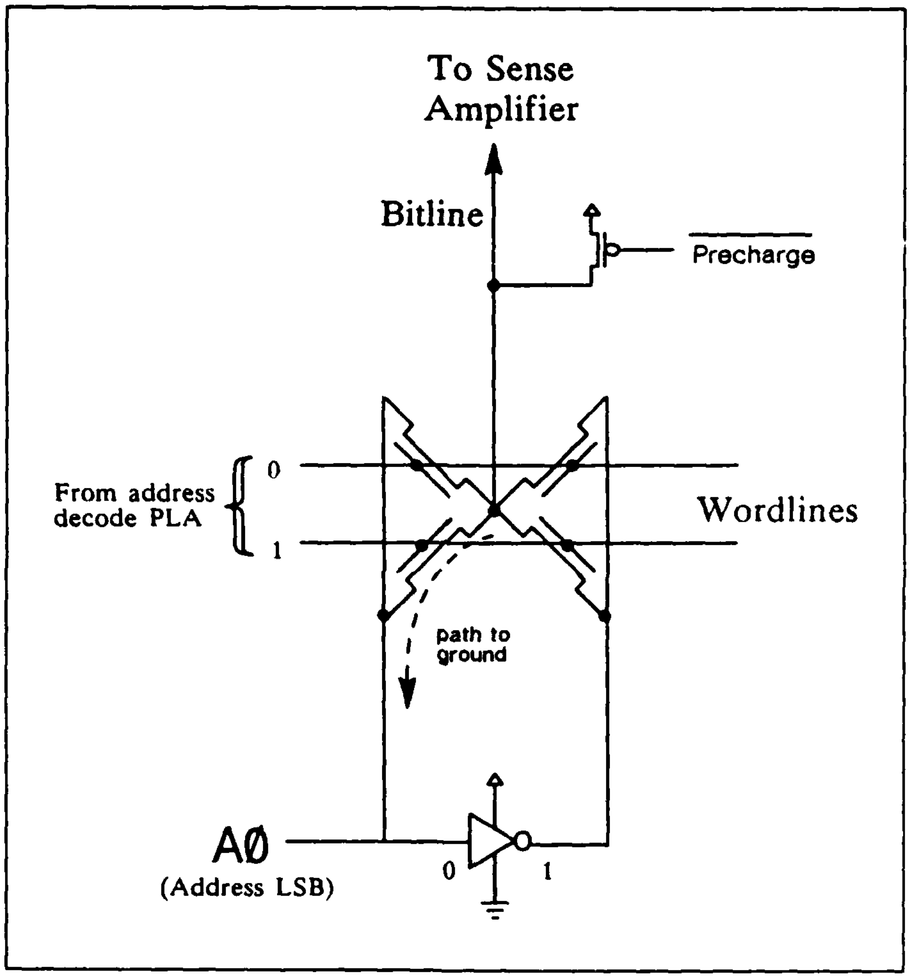


Figure 4.25. XROM Cell [Ros85].

For each '1' in the data there is a transistor on the corresponding bitline which is turned on when the wordline goes high. This allows the precharge on the bit line to bleed off to the low LSB line through the transistor. The resulting low voltage is detected by the sense amplifier and is output as a '1.'

This cell was designed so that it could be laid out by a silicon compiling program. This program lays out the entire ROM, including the sense amps and PLA decoders. Another feature of the program is that it can take the table o. ROM data and optimize the layout to reduce the drain capacitance on the bitlines. This program has been modified to allow it to choose which columns of bits it will re-order in the optimization process.

This last feature is a necessity on the FPASP because the control lines have been optimized for minimum length and minimum routine channel area. This means that only re- arranging of rows must take place within the XROM.

Normally, the XROM optimizer rearranges the columns (and rows) and prints out a mapping of where the columns ended up. This rearrangement of the columns cannot be done in the FPASP because leaving an open channel of busses 64 lines wide on each side of the ROMs for de-scrambling the columns would require too much area. Also, the columns have been selected for each half of the ROM and ordered within each half so that they are as close as possible to their destinations. This reduces the channel area required and also reduces the number of crossovers between the lines.

The only optimizing will be rearranging the rows and inverting the polarity of bitlines which are more than half populated with transistors. This allows the transistors to represent whichever value there is fewer of on that bitline. If there are more '1's than '0's, the transistors are used to represent the '0's instead, and the output of that sense amplifier is inverted.

4.7.6.2 **Laser PROM.** The Laser PROM works on the same principles as the XROM, and uses many of the same cells [Til88]. The LPROM represents data the same way, but now the presence or absence

of the transistor is determined by a laser. Figure 4.26 shows one of the LPROM transistors. The transistor is fabricated with a gap in the diffusion. If the transistor is to be put into the circuit, a laser beam is shined on the gap. Intense local heating causes the diffusion on either side to spread into the gap. When they join, the transistor becomes electrically connected to the bitline.

One of the reasons there is less LPROM storage is that it is eight times less dense than the XROM. This is due to the size of the transistors needed to represent the data. The transistor size is limited by the accuracy of the laser programming equipment available.

The XROM and LPROM supply the same microword bits to the same control lines, and there is no re-arranging of outputs between them. The LPROM must therefore be written so its column assignments are the same as the XROM, since the XROM is fixed.

4.7.7 **Pipeline Registers.** The Pipeline is an array of master-slave flip-flops with muxes at their input which allow them to load from two or three different sources. The bits are sent off

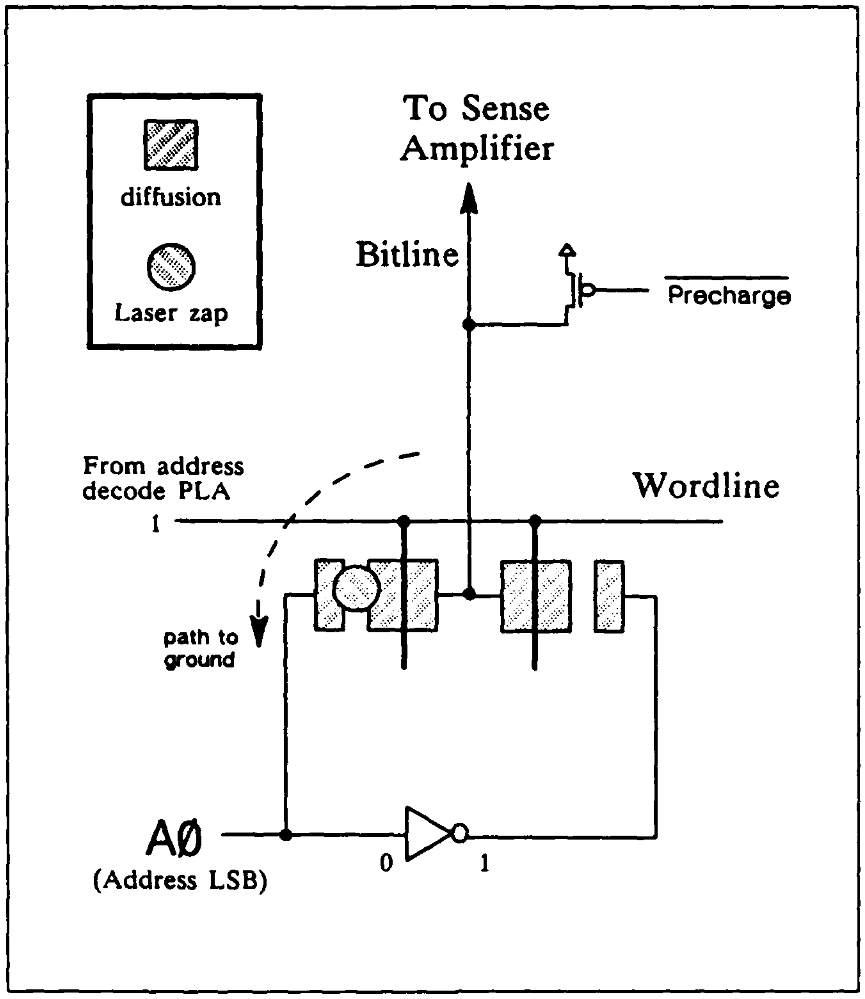
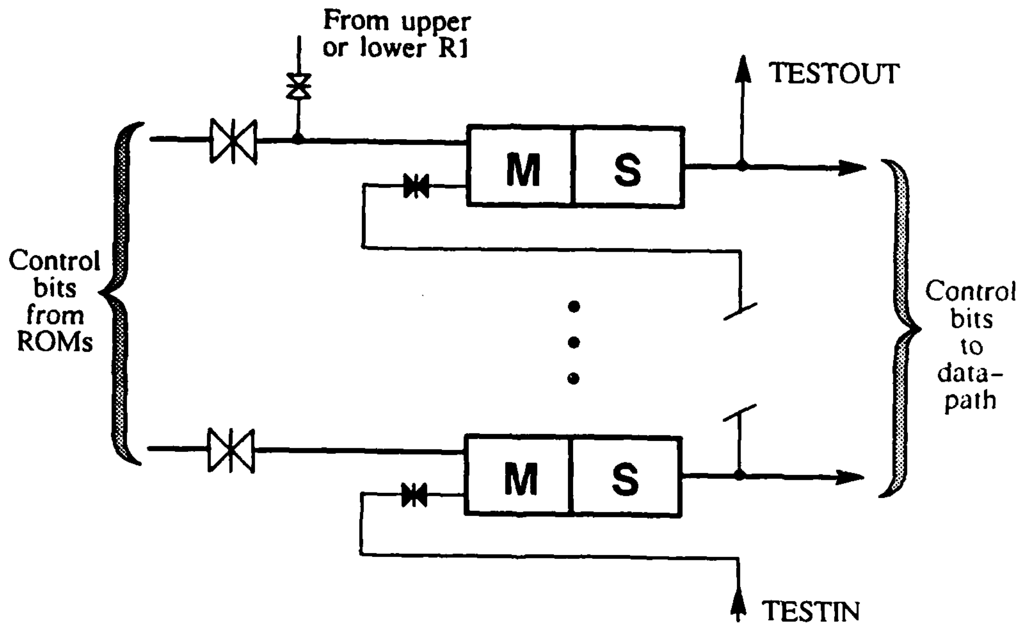


Figure 4.26. LPROM Cell [Til88].

to their respective decoders on the next clock cycle. The outputs of the flip-flops are sized for the control lines they must drive. Long control lines which go to many gates must have more current drive than ones which go only a short distance and end in a few gates.

To cover the various current needs, four different sized stage-up outputs were designed. There are also two possible input circuits, either a two or three input mux. A section of the pipeline showing the two types of input is shown in Figure 4.27. This figure also shows the configuration for the scanpath design-for-testing.



gate open on:

Conrol signal	gate symbol	source of next bit into MSFF
TESTMODE	⧓	bit from previous pipeline MSFF
Macro_Sel bit	⧗	Assembly Language bit from R1
neither	⋈	Normal source from ROMs

Figure 4.27. Pipeline Registers.

The default input for every register is the microword bit out of the ROMs. The other input which they all can choose is the output of the flip-flop next to them. This is for forming the scanpath for testing. The optional input is for taking in a control bit from one of the R1 registers which hold the assembly language control bits.

The pipeline registers give the decoders more time to compute the control signals by presenting the control bits at the start of the clock cycle. If the control bits were to come directly from the ROMs, they would not be valid until after the precharge and access times. This would not do for the bus select decoders, which must have valid control signal before the end of precharge.

4.7.7.1 **Assembly Language Input.** The muxes used by the assembly language are grouped into six sets, each set enabled by one of the control choices in the 'Macrocode Support Mux Selects' field listed in Appendix A. Each routine which carries out an assembly instruction uses one or more of these choices to override the control bits from the ROMs with control bits from the R1s. Table 4.5 shows the bits overridden by each choice. The new control bits are fed in before the pipeline registers so they act just like regular control bits.

Selection	Overridden Control Fields
RSEL	Bus Selects, Bus Ties
BRSEL	Condition MUX Select
SRSEL	E Bus Selects, E bus Tie
ALSEL	ALU/Shifter Control, Bus Selects, Bus Ties
SHSEL	Barrel Shift Amount, Bus Selects, Bus Ties
ISEL	Pointer and Incrementable Register Control

Table 4.5. Assembly Language Override Selections.

4.7.7.2 **Scanpath Hardware.** The scanpath hardware is shown in Figure 4.27. The scanpath design for testing scheme requires special pins for controlling the pipeline muxes and providing input and outputs. Separate pins were used since they were available and the data and address pins all had multiple inputs already.

The first pin is the TESTMODE pin which chains together all of the pipeline muxes to form the scanpath. The other two pins are the TESTIN and TESTOUT pins for passing data to and from the scanpath. Also, the clock lines to the pipeline registers are separate from all the other clocklines in the FPASP. In normal operation these clock lines are connected with the rest of the clock lines externally on the circuit board.

To operate the pipeline, the test clocks are separated from the rest of the clocks, and the **TESTMODE** pin is raised. The scanning can now be done while the rest of the machine holds its state. The control bits can be observed or replaced through the **TESTIN** and **TESTOUT** pin

4.8 Interface Hardware

The FPASP has several pins dedicated to external interfaces. The pad arrangement of the FPASP is shown in Figure 4.28. The pads have been arranged so that they are as close as possible to the part of the circuit they are connected to. For the control pins this is easy to do, but the data and address pins are too numerous and require channel space at the periphery of the circuit to run the busses to them.

The **GO** pin allows the host to reset the FPASP to its starting state. The **GO** signal resets the CAR, stack pointer, and stack tag registers. The FPASP then starts up at address zero of the microcode. This is a small routine which loads the lowest word of the upper memory and checks the LSB. If it is a 1, the code branches to the first word in the LPROM, where the user's routine starts. If the LSB is zero, it continues on into the built-in self-test routine which does an internal check on all the hardware It writes the results out to the external memory.

The **GO** signal must be valid for at least two clock cycles to assure that the microsequencer has been reset. Figure 4.29 shows the timing between host and FPASP, and the states of their external bus drivers.

The **DONE** pin is controlled by a bit from the microcode word. It signals that the FPASP has completed its work and is ready to start again. When the FPASP routine finishes, it branches to a small endless loop which raises the **DONE** line and waits. This bit also puts all of the FPASP output pad drivers into a high impedance state so the host can control the external memories. This can also be done at any time by raising the **HIGHZ** pin.

The two external flag pins **IO1** and **IO2** are direct inputs to the condition mux. There are two pins dedicated for the precharge circuitry. These would normally be connected to Φ_1 clock lines. They take the load off of the lines feeding the logic circuitry, but they can also be used to source precharge separately from the clocks if needed.

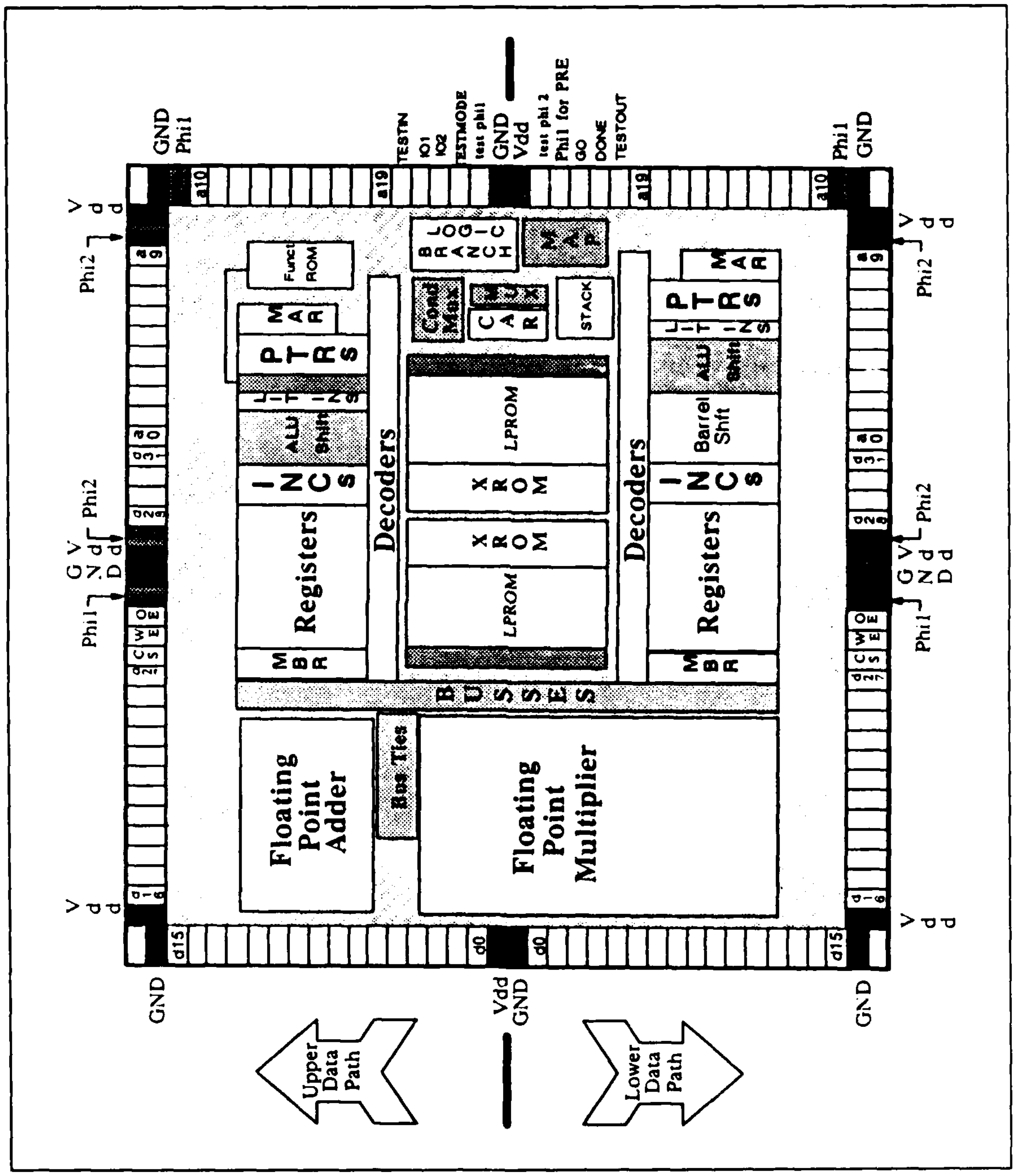


Figure 4.28. Pad Arrangement.

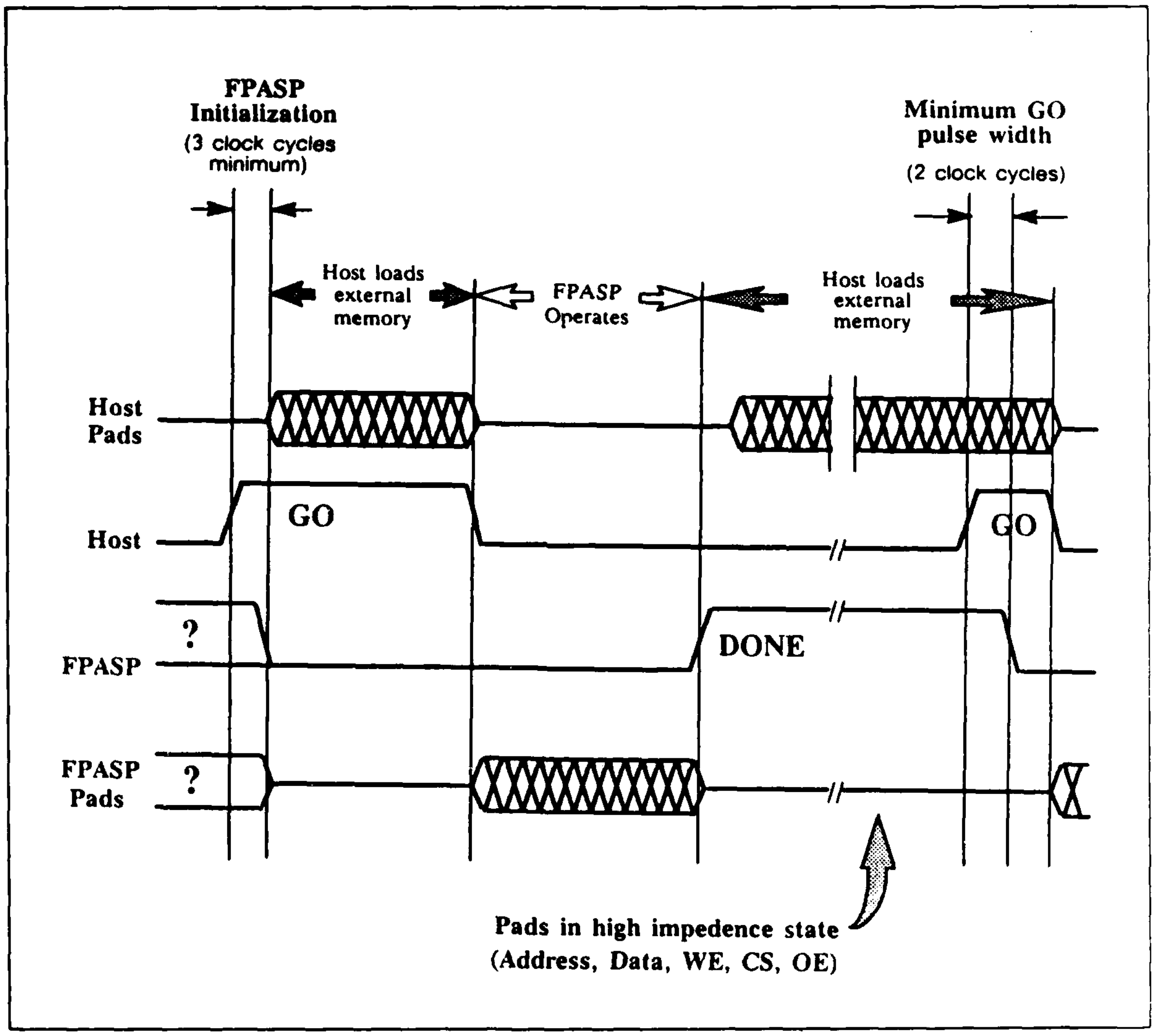


Figure 4.29. Host-FPASP Handshake Timing.

V. Microcell Design and Verification

5.1 Introduction

This chapter presents some of the details that went into the design and verification of the FPASP macrocells. This falls mostly at the CMOS level of Figure 5.1. This figure is embellished slightly from the previous chapters to show where the various CAD tools are applied in the design hierarchy.

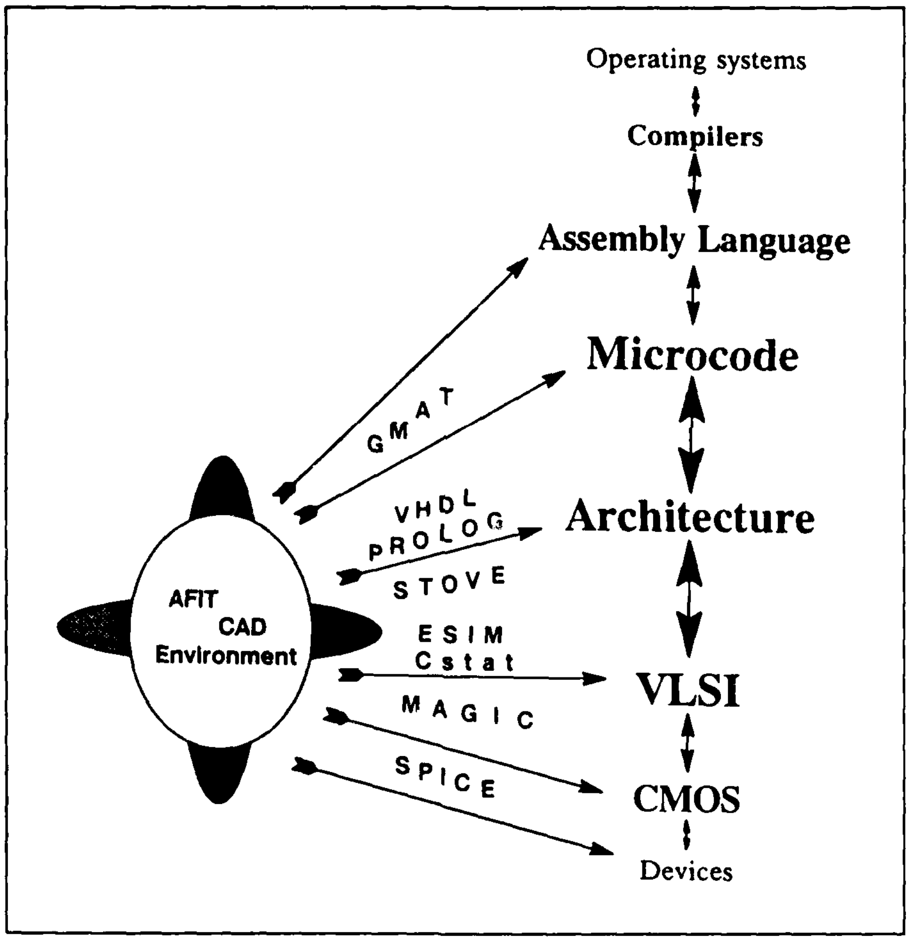


Figure 5.1. CAD Tools Used.

The first part of the chapter will examine the FPASP circuitry from the layout point of view. The next part of the chapter will discuss some of the modeling that was done to verify the design at both the device and macrocell levels. The last part of the chapter presents an outline model of the FPASP written in the VHSIC Hardware Description Language, and shows how that model will be used in the design cycle for FPASP applications.

5.2 Design Style

5.2.1 Cell Libraries.

The hardware design began with an examination of the cell libraries available at AFIT, especially the one created for the ASP chip by Capt. Gallagher. This survey showed that some of the hardware could be used without change, but most of it would need modification. The reasons for redesign were either because the ASP cells were sized for a smaller machine, or the FPASP needed additional functionality from the cells.

The ASP library provided most of the cells for the barrel shifter. That macrocell went together very easily and quickly thanks to the modular design created by Capt. Gallagher. This was true of most of the ASP cells. Using these cells fixed the bus pitch in the FPASP to 81 lambda, but this did not present a problem. Most of the cells which had to be built for the datapath sections were based on parts of ASP cells which already had this pitch.

Another cell library plundered for parts was the Winograd Fourier Transform (WFT) cell library. The pad designs were the closest to what the FPASP needed, and they had been updated for 1.2 micron CMOS. Most of the other WFT cells were for functions not needed in the FPASP.

5.2.2 Standard Cells.

All of the registers are based on the same master-slave flip-flop circuit. The only differences are in the number and type of output drivers, and the number of input multiplexer channels. The various common register circuits are shown in Figure 5.2. This same basic flip-flop is also used without the bus structures for the flag flip-flops. All of the registers conform the 81 lambda pitch of the data paths.

The gates used to build the decoders throughout the FPASP are built from a standard cell library. These cells are derived from the bus select field decoders. The standard cells only take up a little more space than cells custom-designed for each application and they are easier and faster to lay out.

The decoder cells are kept very basic and are made to stack together. The cells contain only transistors, no well contacts or routing lines. After the decode logic is written, the types of gates needed and their arrangement are decided. Then they are laid out cell by cell, leaving room for interconnections. After the cells are laid out the interconnects are made.

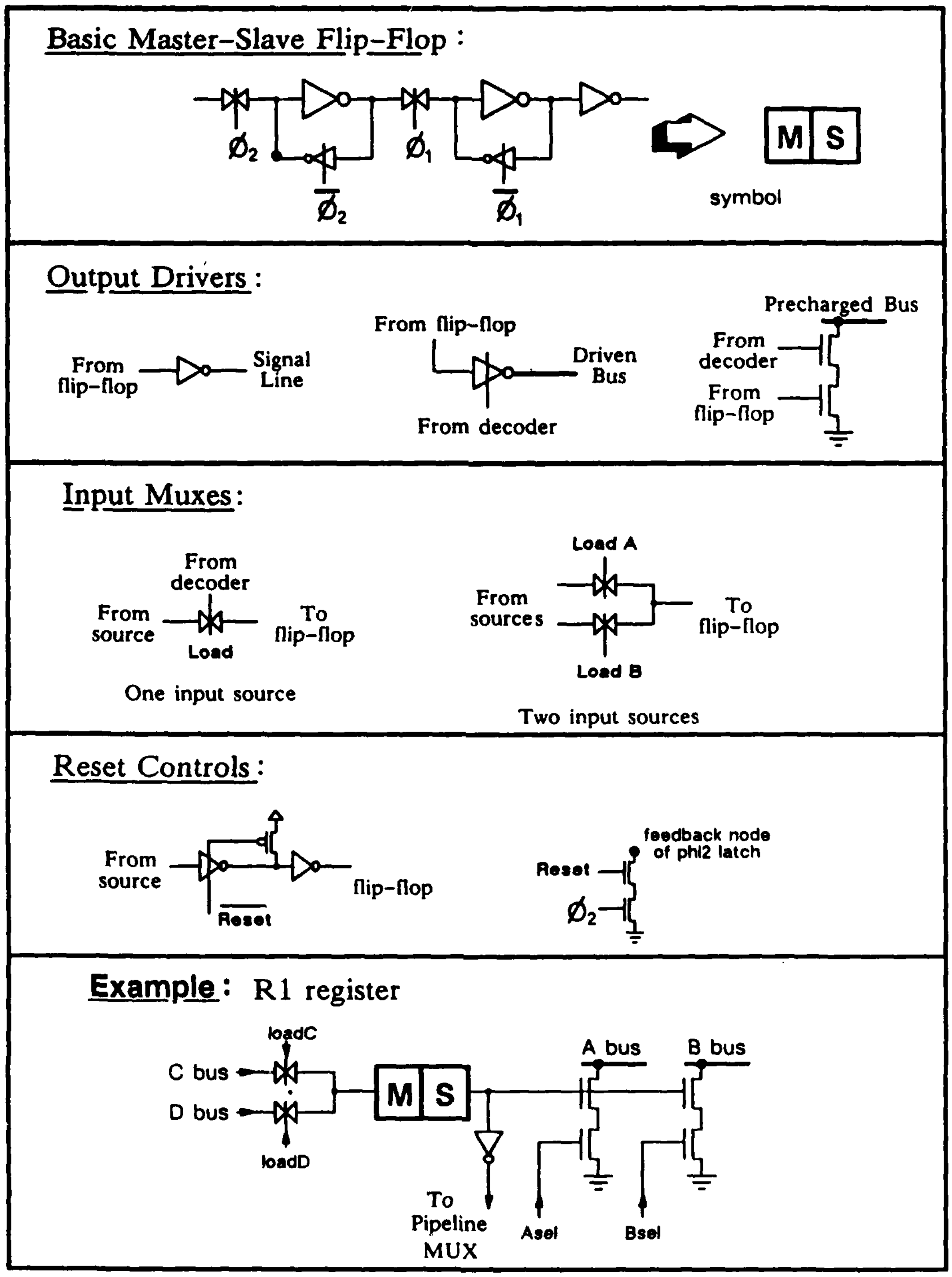


Figure 5.2. Common Register Components.

When an entire decoder is laid out, the hierarchy of cells is flattened and the wells and well contacts are put in. Since the particular fabrication process for the FPASP is unknown, both the N and P wells are drawn in and all have well contacts to help prevent latch-up.

Other standard cells were used for staging up current drive. These consisted of sets of 'dough-nut' inverters. These cells were created before the design of the FPASP was completed, so once the cells were designed the design calculations for current drive were based on these cells.

The most commonly used standard cell library at AFIT is the XROM library. This set of cells can be laid out completely by software [Lin88-2]. It would be difficult to reliably lay out a large ROM. Using the XROM compiler, the process is done correctly in minutes.

5.2.3 **Transistor Sizing.**

The sizes of the various control line drivers depend on how fast that control has to become valid. For example, on controls gated by precharge, all of the decoding circuitry up to the final NAND gate has 10nS to settle. But the circuitry following that gate must be fast because now it is part of the critical timing path.

For the least gate delay, the stage-up of inverter gate widths should be around 2.7 [Gla85]. This rule was followed as much as was practical in the FPASP design. For signals that did not have to become valid immediately after the control bits, the stage-up was stretched to as much as 14. In cases where the timing was critical, SPICE models were simulated to size the transistors.

In the case of the bus ties, they have all of precharge to become valid, so even though the sum of gates in a tie is large, the driver can be small because it has 10nS to charge or discharge those gates. So in this case, a stage-up of 12 was designed.

The ratio of P to N transistor gate widths used in the FPASP ranges between 1.7 and 2. This is to compensate for the higher channel resistance of the P devices. By balancing the resistance of the channels, the amount of current drive for either a high or low output can be kept the same. This makes the rise and fall times nearly equal, so the time the two transistors spend fighting each other at the switchover point is decreased.

Most of the muxes in the ALU/Shifter consist on only N type pass transistors. Since these devices cannot pass a full 5 volt '1', the transistor gate width ratio in the inverter following them must be different from regular inverters to equalize the rise and fall times. For these inverters, the N transistor is made only slightly smaller than the P transistor. This gives it the same channel resistance as the P transistor since it never gets fully turned on.

Transistor sizing is also important for combinational gates. For these circuits, the ratios of the channel resistances were made equal by an analogy to resistors. Putting two transistors in series is the same as putting two resistors in series: the resistance is doubled. Making a transistor's gate wider is like putting resistors in parallel: the resistance decreases. The design goal was to make the total worst case N and P transistor channel resistances the same, within a reasonable layout area.

The NOR type zero flags in the FPASP presented a different design criteria. These are pseudo-NMOS designs, where a P transistor is used to pull the output node high. When any one of the parallel N transistors turns on, it tries to ground the node. In order to make sure the N transistor will pull the node low enough to be seen as a '0', its channel resistance should be at least 4 times less than the P transistor's. For the FPASP, this ratio was 6 to 1. This is because the node is usually in the pulled down state, so the power dissipation is less with the smaller P transistor.

Charge sharing is another problem which is remedied by proper transistor sizing. The loadable registers have two T- gates at their inputs. If the inner T-gate closes and the input changes before the other gate closes, charge gets trapped between them. This charge may be enough to change the state of the register when only the inner gate opens again. To prevent this problem, SPICE simulations were run to size the transistors inside the inner T-gate so their state would not change if the stored charge was applied to their gates.

5.2.4 **Cell Design.** Many cells were designed for the FPASP. From all of this a design style emerged which made cell layout faster. One idea was planning out the placement of all the contacts before the placement of the transistors. The area of the cells is usually limited by how many contacts can be squeezed into the spaces around the transistors.

Time spent planning the contact arrangement made up for time lost pushing circuit elements around to make room for that one last contact that never seems to fit anywhere. In the cell plans, symmetry is the most useful feature to start with. Cells that have symmetrical features are easy to lay out and easy to modify. They also stack well in the macrocells, and make it easier to find misconnected nodes.

Another simple design rule was used at the cell level for deciding which input of a multiple input gate a signal should go to. These gates were usually multiple input NAND gates. These gates have a long chain of N transistors. The charge between the transistors must all be drained to ground to change the state of the output node. The signal which was predicted to arrive last was put on the gate closest to the output node. When that signal arrives, the other transistors are already on, so the only charge not already drained off is the charge of the output node. An example of this is the bus select decoders, where the precharge-bar signal is always the last to arrive.

5.2.5 **Macrocell Design.** The bus select decoders were designed as blank cells. All five of the input control lines run through each cell, as well as precharge. The complements of the inputs are made in the cells where they are used, so only the inverted control bits do not take up wiring channel space. This means that inverters are needed in every cell that requires a complemented input, but the space for the inverters is already there since the cell must be as wide as the register cells anyway.

All of the cells have the same 5 input NAND gate. They are personalized by overlay cells which contain poly lines and vias for connecting the input signal or its complement to the NAND gates. There are 30 small overlay cells for decoding the input control choices 00001 to 11111. This scheme reduces the memory required for the cell library since only the personalizations for each version are stored, rather than entire decoders. The plain decoder cells were also used as the basis for the standard cells used in the other decoder circuits.

Since the bus select signals are common to most of the registers, the decoding circuitry is designed to be the same width as the general purpose register cells, and is located directly next to the register which it controls. This spreads the circuitry out some, but no additional routing space is needed to run the control signals to the registers.

The latter case arose with the barrel shifter control decoder. That decoder was considered for use with the register array since it also decodes five control bits; but three of them would be needed, and although they would take up less area by themselves, the extra area needed to distribute their outputs to the registers and the time needed to lay out those lines made them a poor choice.

The separate-decoder-for-each-register scheme was also used in the ASP, although in that case they were used to support the ASP methodology of making the register array as deep as needed for each application.

The horizontal metal 1 and vertical metal 2 convention common to most AFIT designs was followed in the FPASP whenever possible. This convention makes the most efficient use of the routing channels by reducing the chance the a line will have to cross another of the same type. This convention starts to fail when a large number of contacts are needed. This was especially true in the ALU cells, where routing also had to be run in polysilicon through the bus channel.

Long poly runs were only used where the driver was large and the load was small. This minimizes the RC delay associated with charging up a capacitance through a resistor. In this case, the small load makes the capacitance small, and a 3 lambda width helps lower the resistance of the poly line.

5.3 SPICE Modeling

The SPICE 3 program was used to simulate sections of the circuit at the transistor level [Qua86]. This program provides analog plots of the node voltage throughout the circuit. With SPICE 3, these plots could be viewed on a color monitor for rapid feedback about a design choice. Several optional transistors for the one being tested were kept commented out on adjacent lines in the input file, so changes could be made rapidly. When a good design was reached, a hard copy of the final plot was made.

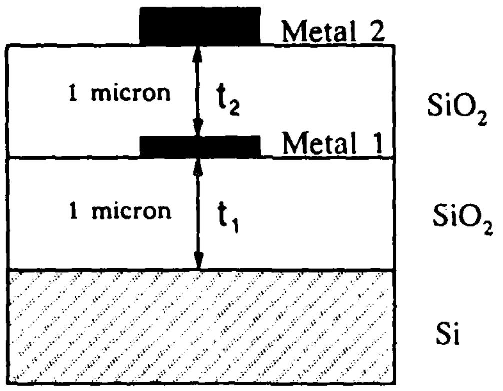
There is no good tool in the AFIT CAD environment for converting the Magic layout designs into SPICE input files. A method for getting exact models from the Magic layout was devised. In Magic, the transistors in question were zoomed in on so they filled the screen, then only the gates, diffusion and diffusion contacts were displayed. On this display, the various node numbers from the model were written using an overhead marker. The various sources and drains were also marked. Then the perimeters and areas of the transistors could easily be transferred from the screen to the proper transistor listing in the model. As each transistor was entered into the model, its display was X'd off with the marker. The marker was also used to draw the boundary between transistors with shared drains.

5.3.1 Register Models.

The FPASP registers are similar to the ASP registers, but have been redesigned for 1.2 micron CMOS based on SPICE simulations. The pulldown transistors have also been sized for the larger busses in the FPASP.

The register model used for this simulation included models for the entire A and B busses, the drains of the other transistors on the busses, and the lengths of the connections within each register on the bus. Values for capacitances given in the Weste text were used, along with the oxide thicknesses shown in Figure 5.3. A factor of two was added for fringing effects that are not accounted for in the flat plate capacitor model used [Wes85].

It turned out that the pulldowns designed for the ASP were large enough for the FPASP, but the flip-flop itself had some designed-in capacitance that was unnecessary for the 1.2 micron process. This was capacitance added to the feedforward inverter of the Φ_2 latch to overcome charge sharing with the node outside of the Φ_2 T-gate. The spice results show that the new design has adequate capacitance without making the inverter's gates longer. This decreases the delay through the inverter, allowing it to latch in a new value faster, and also decreases the time that power is dissipated by the inverter.



$$C = \left(\frac{\epsilon A}{t_n}\right)(2)$$

C = capacitance of the bus (Farads)

A = bus length **x** bus width (microns^2)

$\epsilon = 3.45 \cdot 10^{-11}$ F/m (SiO_2)

t_1 = 1 micron (m1)

t_2 = 2 microns (m2)

Figure 5.3. Bus Model used for SPICE.

The SPICE results also show that an average current of 1mA is needed to keep 32 register cells in the quiescent state of neither loading nor driving. This number is based on a rough estimate derived from the plot of the voltage across a 0.5 ohm resistor put between the circuit and ground.

Also modeled was the current needed to pull down a tied **A** bus line. The peak current during pulldown was 0.8mA. In the worst case, all of the **A** bus lines would be pulled down together; for example, when a pair of registers is being initialized to zero. The peak current for the register pulling all the lines down is 25.6mA. The very worst case for the ground line is when all of the busses (A,B,C,D,E) are pulled down from a **1**, which would require a peak current of 102mA.

At most, only two general purpose register cells at a time can be sinking 0.8ma each on a given ground line per cycle. The cells each contain a 2.4 micron metal1 ground line running through them. Based on a current carrying capacity for metal1 of 1mA per micron of width to prevent metal migration [Wes85], these power supply lines are sufficient for the peak current needs of the register array.

The ASP decoder design was expanded for the FPASP, making use of the SPICE simulation program to verify the operation and speed of the circuits. The model of a decoder was added to the register model used above. This gives the decoder a realistic load. The model uses the worst case delay, where the input to the NAND gate that is farthest from the output node is the last one that changes, which leaves the maximum amount of charge to be drained from the output node to ground. The 10nS precharge time was long enough to cover the worst case delay through the NAND gate, which was only 3.44nS. The delay from the fall of precharge to the rise of the bus select line was 1.8nS. This represents the delay of the precharge-bar NAND gate and the inverters used to stage up the current drive.

5.3.2 **ALU Models.** The worst case delay in the FPASP is the path through the carry select adder. A SPICE model of the longest path through the carry select adder gave a 12nsec delay time. This delay, when added to the delay through the ALU input circuitry and the estimated delay through the shifter stage, gives a total delay of 24 nsec. This includes driving the data and results through the bus ties.

This worst case delay is the main reason for making the width of the precharge pulse as short as possible. A 10 nsec precharge pulse is long enough for the worst case of charging the XROMs, and allows over 25 nsec for the ALU results to become stable on the C bus. The 10 nsec precharge also leaves enough time for the control bits to be decoded and become stable at the inputs to the final prechargebar NAND gates.

5.4 Design Verification

The best looking layout is worthless until it has been verified that it will work correctly, at least at the logic level. The tools available at AFIT for verification include the Esim switch level simulator and several static design checkers.

The first level of check comes during the layout process. The Magic layout tool does continuous design rule checking as the layout proceeds. After the layout is done, the design is converted into a CIF (Caltech Intermediate Format) file by the Magic tool.

The next toll is Mextra, which takes the CIF file and extracts the netlist of transistor connections, plus values for the capacitances of the nodes. The resulting SIM file can be simulated by the Esim program, but first some static checks are done. The Mextra program generates several other files which can point out wiring or other errors.

The most useful of these files is the alias file which lists all of the labels which are electrically connected. The Cstat program recognizes certain labels as being inputs or outputs of the circuit. These labels must be the ones with the fewest characters in them, otherwise Mextra will alias them to another label on the same line, and it will not be visible to Cstat. The solution to this problem is to make sure no other labels are on these lines, or make sure they are longer than the labels put there for Cstat or Esim.

Labels become a mixed blessing with these tools. On one hand, labels make it easy to identify a node, but on the other hand lots of labels are sure to cause problems of the sort mentioned above. In some cases, the labels are in a cell that can't be changed, because it is used in other designs. In any case, the aliases must all be explained or corrected before a simulation can be done.

The static circuit checker Cstat takes the netlist in the SIM file and looks at how all the transistors are connected. It prints a list of all the transistor configurations, and a list of all the transistors which are isolated form the inputs or outputs. Transistors are listed if they cannot be set to 1 or 0, if they cannot be affected by an input, or if the cannot affect the output. Cstat also gives the Magic coordinates for the last gate closest to the node.

This tool was run on a version of the ALU/shifter macrocell. This version contained all of the control and flag circuitry, but the array of ALU cells was shortened down to the four cells which contained unique circuitry. The duplicate cells which were removed were only slowing down the verification process. If the ALU/Shifter works with the cells in the smaller version, then it should work with the full 32 cell array.

The Mextra and Cstat files were used to locate 5 wiring errors. Another error was found while looking for one of the five others. This was a power-tied-to-ground error that was not found by the static checkers. In this case, one of the ALU cells had a P-well contact tied to Vdd. When a line leading to one of the errors was selected in Magic, all of the power lines lit up as well.

After all the Cstat listings are fixed or explained, the next step is to make the circuit Esim-able by removing circuitry that Esim cannot properly simulate. One of these circuits is the sense amplifier in the XROMs. The Fixrom program will replace these sense amps with inverters. The Nofeed program removes feedback clocked inverters, and reports on how many of the clocked inverters were removed. That number should match with the number of flip-flops laid out.

Only after all the above steps have been performed can Esim be used to check the function of the circuit. Esim takes a set of input vectors and runs them through the circuit model. The results are then compared with the expected results. If they do not match, the problem must be deduced and fixed. Then the entire process must be repeated.

The decoders were Esim'd individually and then the smaller version of the ALU was simulated. Several errors were discovered in the decoders: two labels were reversed and the decoder to set the carry flag was outdated. These are the types of errors that a simulation can point out before the overall operation is checked to see if the ALU is actually performing the desired operations

5.5 VHDL Model

A structural model of the FPASP was written in VHDL. This will form the basis of a behavioral model to be written in the future. The tree structure of the model is shown in Figure 5.4. The model will allow microcode to be simulated in software before it is written into the FPASP LPROMs. Testing the microcode is a stage of the FPASP implementation cycle shown in Figure 5.5.

After a working FPASP has been fabricated, copies of it can be tested and then put on the shelf. When a user has an application for it, that application is reduced to an algorithm. The algorithm is then converted to microcode and simulated using the VHDL model. After the code has been verified, the FPASP can be programmed. The programmed FPASP is then placed on a test board and tested again before releasing it to the user.

Two tools are being developed at AFIT to extract structural VHDL models from the SIM netlist files produced by the static circuit checkers. These tools are not yet completed, but the FPASP should provide enough different cells for these programs to prove themselves on. The

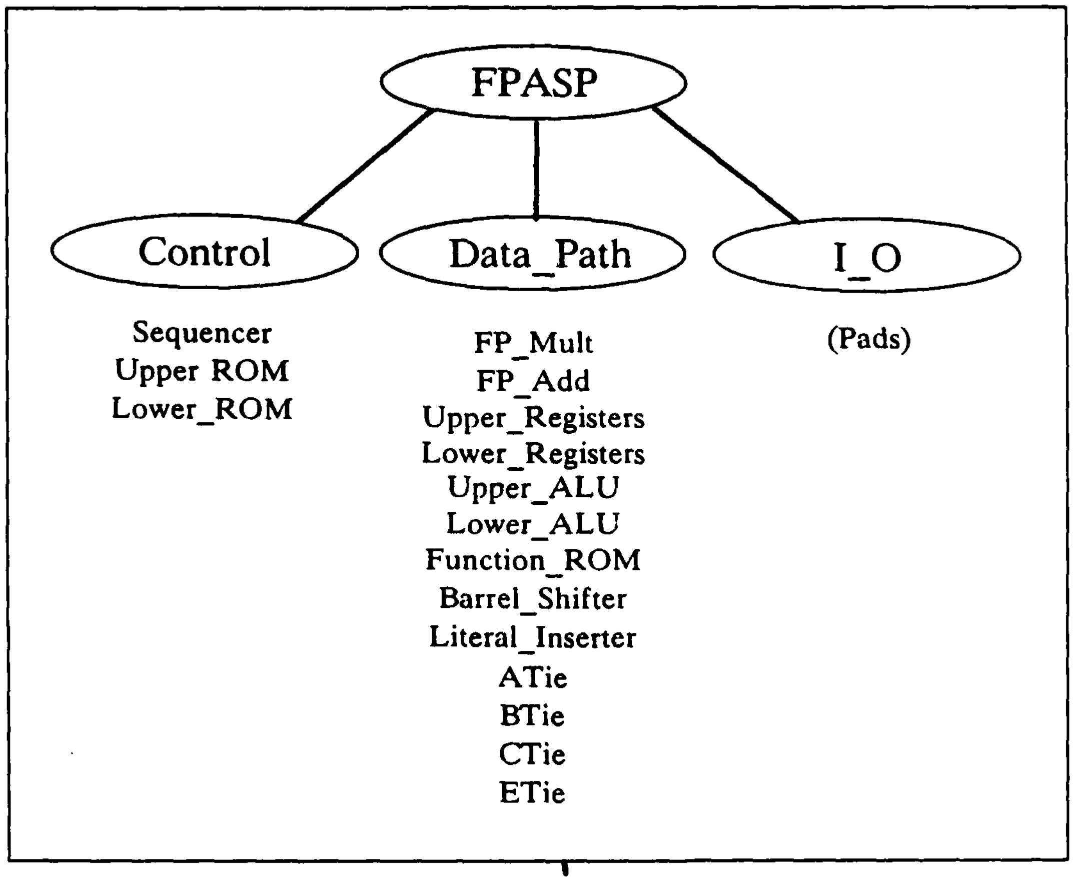
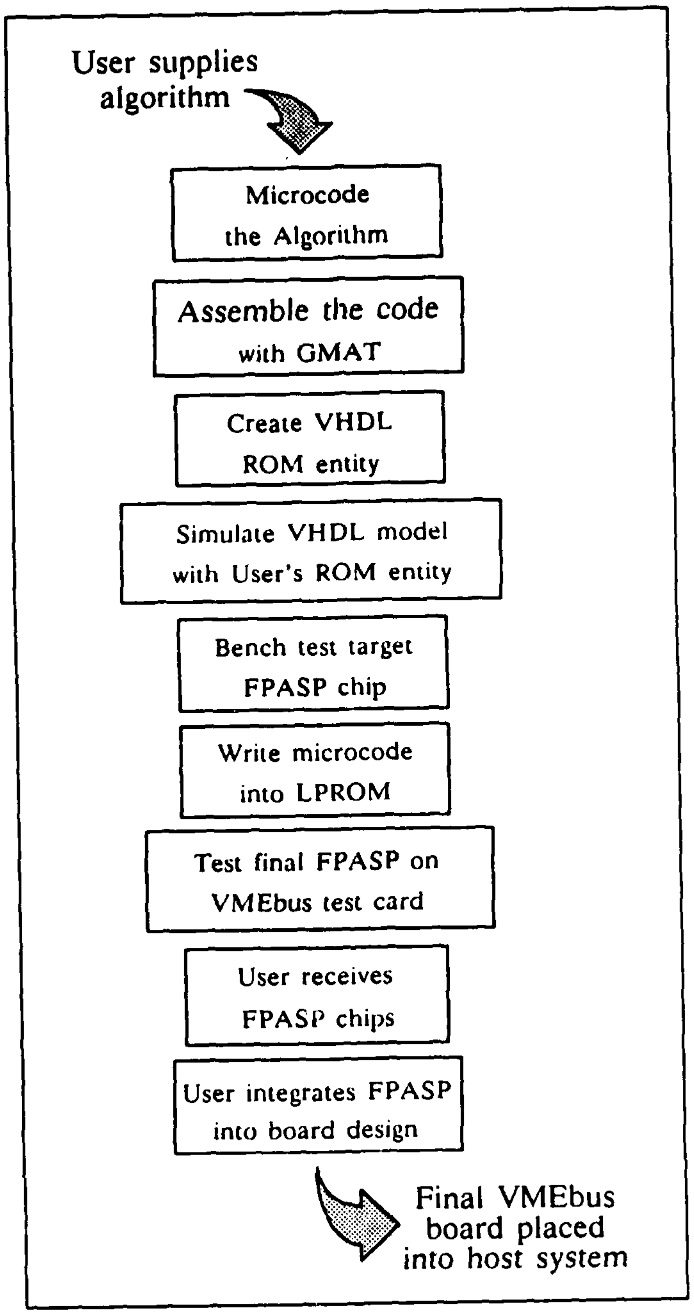


Figure 5.4. VHDL Structural Components.



User supplies algorithm
Microcode the Algorithm
Assemble the code with GMAT
Create VHDL ROM entity
Simulate VHDL model with User's ROM entity
Bench test target FPASP chip
Write microcode into LPROM
Test final FPASP on VMEbus test card
User receives FPASP chips
User integrates FPASP into board design
Final VMEbus board placed into host system

Figure 5.5. FPASP Prototyping Methodology.

STOVE tool is presently extracting such higher level circuits as master-slave flip-flops and pseudo-NMOS Nor gates [Lin85].

The other tool is a Prolog-language extractor. This tool was used at the start of this thesis to reverse engineer some of the cells in the ASP library. It produces a netlist of gates and miscellaneous transistors. The circuit logic can then be drawn out by hand [Duk88].

VI. Results

6.1 Introduction

Figure 6.1 shows where this chapter will concentrate on in the hierarchy of processor design. Mostly this will be microcode results from the EE588 class and routines written to support those projects. It will also give some statistics on the FPASP chip and compare some of the FPASP's architectural features with other 32 bit processor architectures.

The FPASP thesis effort took a total of 900 hours. Table 6.1 shows the approximate percentage of time spent on various topics.

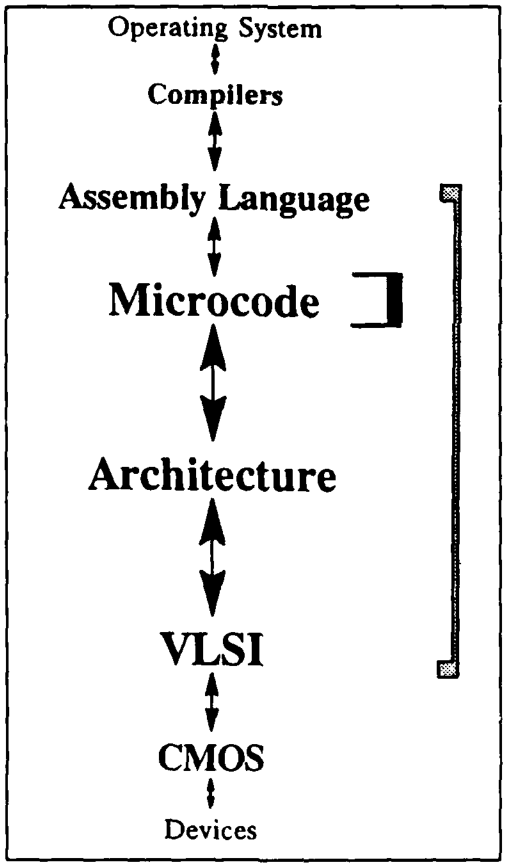


Figure 6.1. Theme Figure.

Topic	*Percent Time*
Architecture	35%
VLSI Design	35%
Microcode	10%
Layout (up to this printing)	10%
Documentation for EE588 Class	8%
VHDL coding	2%
TOTAL	100%

Table 6.1. Areas of Effort

Macrocell	*Percent Area*
Floating Point Multiplier	27%
Floating Point Adder	7%
Other Processing H/W	4%
ROMs	11%
Registers	16%
Control Section	2%
Bus channels	11%
Padframe	19%
Miscellaneous Hardware	3%
TOTAL	100%

Table 6.2. FPASP Chip Area Allocations

6.2 VLSI Layout Statistics

The FPASP is to be fabricated in a 1.2 micron double-metal CMOS process. The final package will be a 144 pin-grid-array. Table 6.2 lists the percentage of the total die area to be taken up by each of the main sections of the FPASP. The die size is 315 mil by 380 mil, for a total of 119,700 mil^2. This is under the proposed limit of (350 mil)^2. The areas are based on macrocells already laid out and the estimates used for the floorplan. The projected transistor count for the FPASP is around 186,000. The multiplier alone has over 75,000 transistors. These are mostly in the array of 1364 full adders and 327 half-adders which calculate the mantissa.

6.3 Microcode Projects Summary

6.3.1 Support Routines. The routines summarized in Table 6.3 were written for the EE588 class to call in their routines. They are part of the library of routines to be included in the FPASP XROM. They include basic routines for matrix algebra and the Newton-Raphson inversion routine listed in Appendix B. Table 6.3 lists the number of lines, the number of cycles needed to run, and the number of floating point operations performed (FLOPS).

Routine	*lines*	*clock cycles*	*FLOPS*
Dot Product	8	2n+6	2n-1
Matrix x Vector	5	11m+2nm-3	2nm-m
Matrix x Matrix	5	2p+11pm+2pnm-1	2pnm-pm
For two matrices mxn times nxp			
Vector Scaling	5	3n+4	n
For vectors of n elements			
Newton-Raphson Invert	15	31	12

Table 6.3. Subroutine Statistics

The number of clock cycles depends on the size of the input matrix or vector. The matrix algebra routines can operate on inputs of any size, limited only by the size of the external memory. The matrix-matrix multiply routine calls the matrix-vector multiply routine, which calls dot-product. The efficiency of each routine in terms of floating point hardware usage can be calculated as FLOPS per clock cycle. Using this metric, the efficiency of the dot-product routine approaches 100% as the number of elements in the vectors increase. The Newton-Raphson routine runs for four iterations, which is enough to have the routine to converge on the final result. Its floating point efficiency is 12/31, or 39%.

6.3.2 EE588 Projects Summary. This chapter summarizes the results of the EE588 microcode projects [Lin88]. Table 6.4 lists the algorithms implemented by these projects, the amount of code each one took up (excluding the routines listed in the previous section), and how efficiently they used the floating point hardware.

The sizing results show clearly that the FPASP ROMs are large enough for complex algorithms such as Kalman filtering or neural net back-propagation. The ROMs can hold these routines, the subroutines they call, and a self-test routine. These routines were written by students, many of whom had not written microcode before. Therefore, these results do not represent the ultimate performance capabilities of the FPASP. A better measure would be the routines listed in Table 6.3. For example, the efficiency of the Kalman filtering routine can be increased by correcting a programming flaw.

The efficiency of floating point hardware use is an important result, since those pieces of hardware have been given the most area on the chip. If the efficiency was always low it would indicate that the FPASP architecture was not refined enough for its intended application. The

Project	*lines*	*% use of FP*	*MFLOPS/sec*
Kalman Filter	406	35%	8.8
Neural Nets			
forward propagation	130	76%	18.9
backward propagation		8%	2
LMS Adaptive Filter	81	51%	12.5
Log Base 2	19	70%	17.5
Coordinate Transformation			
rectangular to spherical	23	16%	4
spherical to rectangular	14	7.2%	1.8
Arctangent	47	6.2%	1.6
Singular Value Decomposition	97	36%	9
Newton-Raphson Square Root	23	35%	8.8

Table 6.4. 588 Project Summary

most expensive hardware should be getting as much use as possible, otherwise those functions might be better off done by a separate device.

Low efficiency could have been due to many causes. The architecture may have lacked the flexibility or processing elements needed to keep the floating point hardware supplied with data. The algorithm may not have required intense floating point calculations, in which case the FPASP would be the wrong processor to use. The architecture may have lacked the support needed for efficient microcode programming techniques, or the support may not have been used by the programmer.

The latter case can be seen in some of the EE588 projects, where this was the first exposure to microcode for most of those students. The FPASP is a complex machine to be learning microcode on, and the version they used was not the final version, since one of the purposes of the projects was to refine the design.

As it turned out, the FPASP met the objectives of being generic enough for a variety of applications, and flexible enough to perform them efficiently. These results indicate which types of algorithms are best for the FPASP. The ones which use the features of the FPASP architecture for indexing into matrices and transferring the data into and out of the floating point hardware with a minimum of contention for the D bus are the best. The routines which do mostly integer manipulations to set up a floating point operation are the poorest ones by this efficiency metric.

A self test routine was also written. It used 370 lines of code. This reflects the problem with self-test: a large subset of the possible commands must be run through to fully test the machine. The test code was written to detect stuck-at faults, and assumes the hardware will function properly if fabricated properly.

The following section lists some design changes made, based on the results of the EE588 projects. Application of some of these changes to the self-test code could reduce it by 90 lines of code. This frees up more XROM storage for the library of common subroutines.

6.3.3 Result of Project Suggestions. As part of the project write-ups, the students were asked to suggest modifications which would make their code more efficient. Their suggestions are listed in Table 6.5. The bottom of the table lists the ones which could be added to the FPASP architecture without major modifications. Of the ones not added, one has been implemented in a round-about way. The suggestion to have the register selections indexed in addition to being directly selectable from the ROM can be done using the hardware added for the assembly language.

As a result of adding that hardware, the register select fields can be overwritten from the R1 registers using the RSEL command. The user can format the data in the R1s to select any set of registers and bus ties desired. The register selections can be incremented or decremented using the ALUs, as long as an operation in one field does not overflow into the next one.

The other suggestions require too many control bits. The registers for swapping could easily be built in pairs using the pointer registers, but there are no control bits left free to control the swapping. The control would have to be stuck into one of the unused choices in an existing field, which would mean it could not be done concurrently with the other instructions in that field.

6.4 Architectural Comparisons

The FPASP architecture shares features with other processors. How these features affect the software are seen in the code written for the FPASP as well as in the literature. The most common shared feature is the pipelined microsequencer control and pipelined floating point hardware.

For example, the Motorola MC88100 has a pair of floating point pipelines [Mot88]. The multiplier is six levels deep, whereas the FPASP has only two levels. The tradeoff made here is speed versus area. The Motorola chip takes longer to do a multiply, but has less redundant hardware. The FPASP multiplier uses redundant hardware to produce a result every two clock cycles.

The result is the FPASP process is much simpler to keep track of both in the microcode and in the hardware. The MC88100 requires extra registers to keep track of the state of the machine at each level of the pipeline. This is in case the contents are invalidated by a subsequent operation and the pipeline must be flushed.

Project	*Suggested Improvements*
Kalman Filter	literal insert on both datapaths two registers load at once from C bus C bus tie Registers load from A and B busses
Neural Nets	literal insert on both datapaths
LMS adaptive filter	indirect register selection incrementable MAR separate literal and next address fields
Log Base 2	none
Coordinate Transformation	more registers load from D bus register swaps
Singular Value Decomposition	literal insert on both datapaths incrementable MAR
Self Test	multiple registers load from C bus indirect register selection register swaps
Changes made to FPASP	literal insert on both datapaths incrementable MAR three registers load from D bus C bus tie

Table 6.5. Changes Suggested by 588 Students

In the FPASP there is only one stop in the pipeline, but two clock cycles are used to let the results settle. The multiplier has no internal registers to reset. If a multiply in progress is invalidated there is no penalty for flushing the pipeline. The next operation can be loaded before the invalid one is done and the hardware will still settle on the correct result after two clock cycles.

The tradeoff of hardware for speed in the FPASP is not made in the MC88100 because of its more general purpose intentions. Also, the two cycle delay for the FPASP multiplier can be taken care of in its microcode. This shows where the two architectures are fundamentally different despite some shared features.

6.4.1 No 'RISC' Involved. The MC88100 is a 'RISC' (Reduced Instruction Set Computer) type of processor which uses simple commands that must be decoded in a single clock cycle. It does not have microcoded routines to keep track of the multiplier's latency. Each command must either be carried out immediately or pushed into a pipeline.

Although the FPASP has a 'reduced' set of instructions, they are reduced only in the sense that they represent a lower level of hardware. The number of possible combinations is immense, and the routines written in microcode are basically single, very complex instructions.

Even if the FPASP was only running programs written in its assembly language it would not qualify as a 'RISC' despite the fact that there are only 30 assembly 'instructions'. H. M. Sprunt has proposed six criteria for a machine to qualify as an 'RISC' [Mit86]; they are listed in Table 6.6.

1. *Single-cycle operation.* This facilitates the rapid execution of simple functions which predominate a computer's instruction stream and it promotes a low interpretive overhead.
2. *Load/store design.* Follows from a desire for single-cycle operation.
3. *Hardwired control.* For the fastest possible single-cycle operation. Microcode leads to slower control paths and adds to interpretive overhead.
4. *Relatively few instructions and addressing modes* This facilitates a fast, simple interpretation by the control engine.
5. *Simple instruction format.* The consistent use of a simple format eases the hardwired decoding of instructions, which again speeds control paths.
6. *More compile time effort.* RISC machines are predicated on running only compiled code. This offers an opportunity explicitly to move static runtime complexity into the compiler.

Table 6.6. Suggested RISC Criteria [Mit86].

The FPASP certainly meets the last three of these criteria, but the fact that it is microcode driven takes it out of the realm of 'RISC' processors. This puts it with the 'CISC' processors (Complex Instruction Set Computers).

Although 'complex' usually means many instructions, the FPASP has only 30. But over half of these can be programmed by the user, making them as complex as needed to simplify the code for their routine. So a single complex FPASP instruction could do a multiply-and-accumulate, or memory management chore for an operating system.

The pre-defined instructions can be considered complex also, since they can be used to implement nearly all of the microcode instructions. They represent a complex set of possible instructions.

VII. Conclusions and Recommendations

7.1 Conclusions

The FPASP research effort has pointed out one of the strengths of the AFIT VLSI program. This is the use of ongoing research for class projects. The researcher gets additional data and the students get to work on a real project instead of a dry textbook example. This also provides continuity, since some of those students will be picking up this research next year. The researcher also gets experience producing documentation suitable for students who are unfamiliar with the research. The quality of the results and the experience gained by both sides benefit everyone.

Projects with the scope of the FPASP require this type of synergism in order to produce meaningful research. If this research had been attempted without the class project the result would surely have been less satisfactory. This conclusion is even more meaningful now that the next class of students have selected their theses, and the FPASP is being pursued for two of them.

The FPASP presents a new rapid prototyping methodology, but this in no way invalidates the one proposed by Capt. Gallagher. In fact, since most of the cells in the FPASP have been designed to the same bus pitch as those already in the ASP library, they can also be drawn on for prototyping those types of ASPs. The new cells extend the scope of that library into the double precision and laser-programming arenas.

The dual 32-bit processor architecture of the FPASP is a good way to incorporate double precision hardware into a machine which must also perform integer operations. The 32-bit datapaths match those of other processors in common use. This will make interfacing to other processors simpler, whether indirectly through a memory bank or directly through a bus. Any other scheme for splitting a group of 64 bits into a set of smaller groups would either be asymmetric, and therefore harder to lay out, or it would require more separate datapaths, none of which would be a convenient number of bits wide. More separate datapaths would also require more pins if each were to have its own memory bank and address bus.

The choice of six icrementable registers and four pointer registers is enough for most applications involving matrix algebra. Nested loops did not go beyond three, so incrementers were available for loops both in the subroutines and in the main routines. Most of the matrix algebra required only three matrices: two sources and one result.

The depth of the external stack was not reached by any of the routines written by the EE588 students. The deepest use was 53, but this was for a routine that only called itself. If the routine called other subroutines as well as calling itself, the deeper stack would be more critical.

7.2 Recommendations

In keeping with the conclusions made above, a possible class project presents itself for the FPASP VHDL model. The existing model is only a frame for organizing the behavioral descriptions of the components. This makes it ideal for a class project. All of the groups will be working on a common machine, but there are enough components to make each project worthwhile. The structural frame provides the means to define the scope of each group project, and it specifies all of the external signals.

The FPASP should be designed with LPROM cells which use metal links for programming rather than diffusion links. The laser programming hardware presently available is already capable of cutting metal lines with the required accuracy. Forming links in diffusion apparently requires more precise control of the laser pulse's frequency and duration than cutting metal links does. The fusable metal link requires about the same amount of space as the diffusion link, so there is no reason to rely on the riskier diffusion process at this time. All of the software and hardware available now can be used for either process [Til88].

The previous ASP prototyping method proposed by Capt. Gallagher should be retained. It could even be used for classes in advanced VLSI design, where the students are given an application and given this library of cells to implement it.

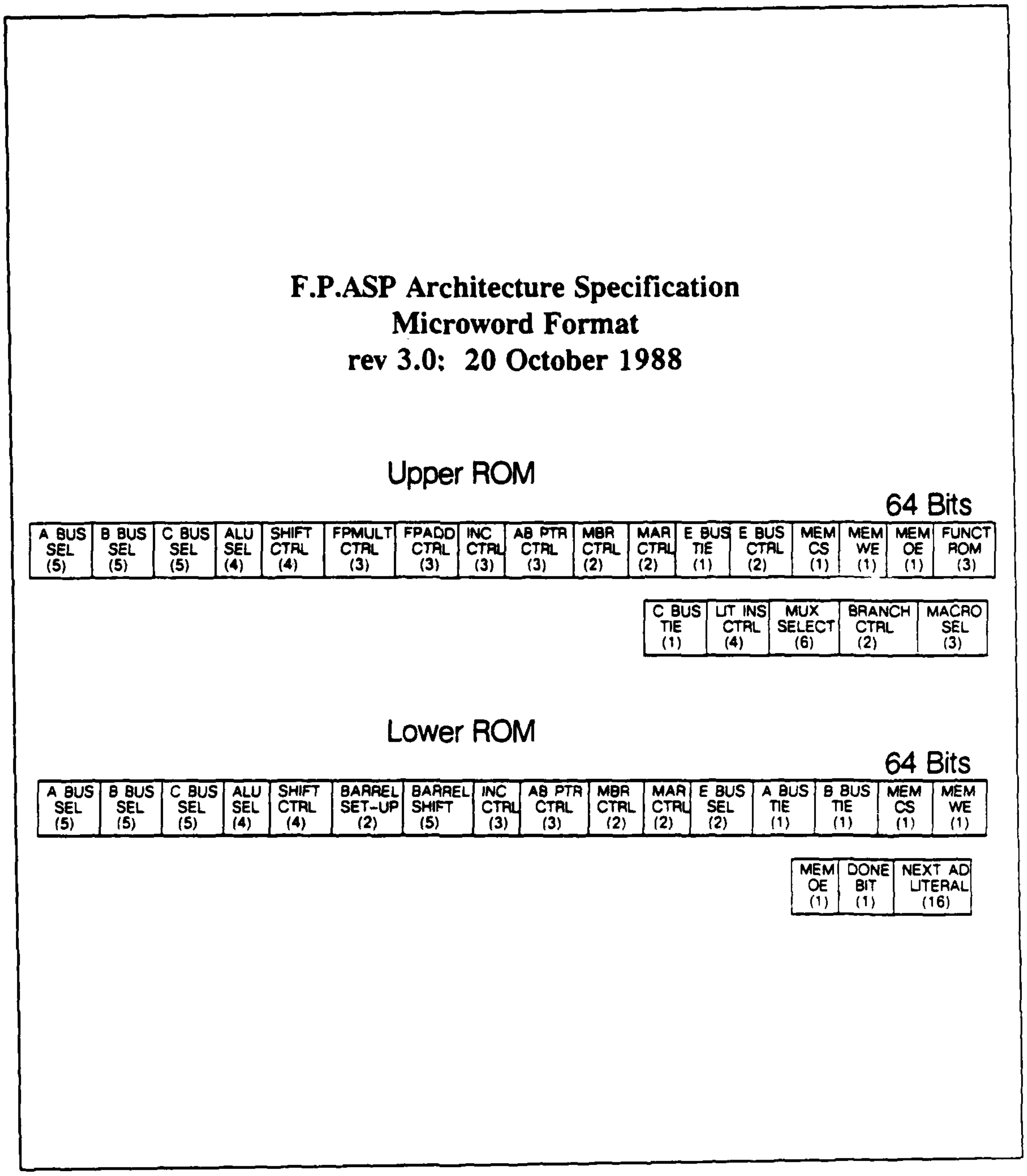
The core processor of the FPASP could also serve as a nucleus for laser programmable ASPs in other areas with computationally intense applications. There are many areas where all of the necessary hardware could be integrated onto the same chip as the processor. These would re-use the FPASP layout and merely replace the floating point macrocells with ones more suitable for the area in question.

For example, the multiplier and adder could be replaced with macrocells for manipulating data in byte-size chunks, along with error detection and correction circuitry. This would make the new ASP ideal for a variety of communications applications. Or the floating point hardware could be sacrificed for slower but smaller versions. The space saved could then be used for hardware to interconnect a large array of ASPs much like the INMOS 'Transputer'. The possibilities are almost endless once the basic core is designed.

One recommendation for the FPASP in particular is to send out some parts for fabrication as MOSIS tinychips before the final chip is fabricated. This has already been done in the case of the LPROM and the hardware on the ASP chip. The pads are the derived from the WFT chips, so those can also be tested.

Some circuits which would benefit from more testing are the ALU/Shifters and two types of incrementable registers. These represent two of the longest delay paths and should be fully tested. In the case of the ALU, a shortened version like the one used for ESIM could be fit on a forty-pin tinychip. This would provide enough pins for all the controls (8), flags (5) and data I/O (12), with others left over for observability of the control decoders.

The programmable assembly language idea could be pursued farther. If the mapping of the R1 registers to the macromuxes on the control pipeline was partially laser-programmable, the user could define instruction formats as well as the opcode and microcode routine. This could be useful if a compiler were written for the FPASP. The 'specific application' would then be support for the compiler.



F.P.ASP Architecture Specification
Microword Format
rev 3.0: 20 October 1988

Upper ROM

64 Bits

A BUS SEL	B BUS SEL	C BUS SEL	ALU SEL	SHIFT CTRL	FPMULT CTRL	FPADD CTRL	INC CTRL	AB PTR CTRL	MBR CTRL	MAR CTRL	E BUS TIE	E BUS CTRL	MEM CS	MEM WE	MEM OE	FUNCT ROM
(5)	(5)	(5)	(4)	(4)	(3)	(3)	(3)	(3)	(2)	(2)	(1)	(2)	(1)	(1)	(1)	(3)

C BUS TIE	LIT INS CTRL	MUX SELECT	BRANCH CTRL	MACRO SEL
(1)	(4)	(6)	(2)	(3)

Lower ROM

64 Bits

A BUS SEL	B BUS SEL	C BUS SEL	ALU SEL	SHIFT CTRL	BARREL SET-UP	BARREL SHIFT	INC CTRL	AB PTR CTRL	MBR CTRL	MAR CTRL	E BUS SEL	A BUS TIE	B BUS TIE	MEM CS	MEM WE
(5)	(5)	(5)	(4)	(4)	(2)	(5)	(3)	(3)	(2)	(2)	(2)	(1)	(1)	(1)	(1)

MEM OE	DONE BIT	NEXT AD LITERAL
(1)	(1)	(16)

A.1 Definitions of Microcode Fields

```
        Revision 3.0   10 November 1988


Upper ROM Fields  -  64 bits

bits 0-4     Upper A Bus Select

00000:      NOP1   (no drive)  (DEFAULT)
00001:      AU=R1  (upper A bus = upper R1)
00010:      AU=R2
                |
11001:      AU=R25
11010:      AU=IN1    (upper A bus = upper INC1)
11011:      AU=IN2
11100:      AU=IN3
11101:      AU=APT    (upper A bus = A pointer)
11110:      AU=BPT
11111:      AU=MBR    (upper A bus = upper MBR)

bits 5-9     Upper B Bus Select

00000:      NOP2
00001:      BU=R1
00010:      BU=R2
                |
11001:      BU=R25
11010:      BU=IN1
11011:      BU=IN2
11100:      BU=IN3
11101:      BU=FP*      (upper B = result of FP MULTIPLIER, MSB's)
11110:      BU=FP+      (upper B = result of FP ADDER, MSB's)
11111:      BU=MBR

bits 10-14   Upper C Bus Select

00000:      NOP3    (no load)  (DEFAULT)
00001:      R1=CU   (upper R1 loads from upper C bus)
00010:      R2=CU
                |
11001:      R25=CU
11010:      IN1=CU  (upper INC1 loads from upper C bus)
11011:      IN2=CU
11100:      IN3=CU
11101:      APT=CU  (A pointer loads from upper C bus)
11110:      BPT=CU
11111:      MBR=CU  (Upper MBR loads from upper C bus)

bits 15-18     Upper ALU Select
```

0000:	NOP4	
0000:	MOVUN	(SHIFTER_INPUT=A, flags unaffected)(DEFAULT)
0001:	ORU	(SHIFTER_INPUT = A or B)
0010:	ANDU	(SHIFTER_INPUT = A and B)
0011:	XORU	(SHIFTER_INPUT = A xor B)
0100:	MOVU	(SHIFTER_INPUT = A, affects flags)
0101:	NANDU	(SHIFTER_INPUT = A nand B)
0110:	NORU	(SHIFTER_INPUT = A nor B)
0111:	NOTU	(SHIFTER_INPUT = A')
1000:	INCU	(SHIFTER_INPUT = A + 1)
1001:	SETU	(SET CARRY FLAG FLIP_FLOP)
1010:	ADCU	(SHIFTER_INPUT = A + B + cy flip-flop)
1011:	ADDU	(SHIFTER_INPUT = A + B)
1100:	NEGAU	(SHIFTER_INPUT = -A)
1101:	SUBU	(SHIFTER_INPUT = A - B)
1110:	SWBU	(SHIFTER_INPUT = A - B - cy flip-flop)
1111:	DECU	(SHIFTER_INPUT = A -1)

bits 19-22   Upper Shift Control (INPUTS COME DIRECTLY FROM ALU)

0000:	NOP5	(SHIFTER DOES NOT DRIVE C BUS) (DEFAULT)
0001:	GNDCU	(C = 0 short C bus to GND, shift flags unaffected)
0010:	PASSU	(C = SHIFTER INPUT, shift flags unaffected)
0011:	SLOTU	(SHL with SH_OUT into LSB)
0100:	SLMSU	(circulate left with MSB into LSB)
0101:	SLCYU	(SHL with CY out of ALU into LSB)
0110:	SLOU	(SHL with 0 into LSB)
0111:	SL1U	(SHL with 1 into LSB)
1000:	SRLSU	(circulate right with ALU LSB into MSB)
1001:	SRCFU	(SHR with carry flip-flop into MSB)
1010:	SRSU	(SHR with SIGN FF into MSB)
1011:	SROTU	(SHR with SH_OUT FF into MSB)
1100:	SRSEU	(SHR with sign extension = MSB into MSB)
1101:	SRCYU	(SHR with CY out of ALU into MSB)
1110:	SROU	(SHR with 0 into MSB)
1111:	SR1U	(SHR with 1 into MSB)

bits 23-25   Floating Point Multiplier Control

000:	NOP6	
000:	FP*	(do floating point multiply) (DEFAULT)
001:	FP*D	(drive results onto C bus, latch flags)
010:	FP*L	(load A,B bus inputs)
011:	FP*LD	(load A,B bus inputs, drive C bus, latch flags)
100:	INT*	(do integer mult: this must be selected during INT*)
101:	INT*D	
110:	INT*L	(load 32 bit integers from A & B busses)
111:	INT*LD	

bits 26-28 Floating Point Adder Control

000:	NOP7	
000:	FP+	(do floating point add) (DEFAULT)
001:	FP+D	(drive results onto C bus, latch flags)
010:	FP+L	(load A,B bus inputs)
011:	FP+LD	(load A,B bus inputs, drive C bus, latch flags)
100:	FP-	(do fl pt subtr: this must be selected during subtr)
101:	FP-D	(drive C bus while doing subtract)
110:	FP-L	(load new inputs while doing subtract)
111:	FP-LD	(load and drive while doing subtract)

bits 29-31 Upper Incrementable Registers Control

000:	NOP8	(no increment) (DEFAULT)
001:	UIN1+	(increment upper INC1)
010:	UIN2+	(increment upper INC2)
100:	UIN3+	(increment upper INC3)

bits 32-34 A,B Pointer Controls

000:	NOP9	(no change) (DEFAULT)
001:	AIN_L	(load A increment size)
010:	BIN_L	(load B increment size)
011:	ABIN_L	(load A,B increment sizes)
100:	APT+	(increment A pointer)
101:	A+_AL	(increment A pointer, load A increment)
110:	B+_BL	(increment B pointer, load B increment)
111:	BPT+	(increment B pointer)

bits 35-36 Upper Memory Buffer Registers Control

00:	NOP10	(no action) (DEFAULT)
01:	R1=DU	(Load upper R1 from upper D bus)
10:	R2=DU	(Load upper R2 from upper D bus)
11:	MB=DU	(Load upper MBR from upper D bus)

bits 37-38 Upper Memory Address Register Control

00:	MARU	(upper MAR drives upper Address bus) (DEFAULT)
01:	MAR=CU	(Drive upper addr bus & load from C bus)
10:	MAR=EU	(Drive upper addr bus & load from E bus)
11:	NMARU	(upper MAR does not drive upper addr bus, Data and memory pads go to high impedence)

bit 39 E Bus Tie

00:	NOP11	(no drive) (DEFAULT)
01:	ETIE	(tie upper and lower E busses together)

bits 40-41 Upper E Bus Select

00:	NOP12	(no drive) (DEFAULT)

01:	E=APT	(drive A pointer onto upper E bus)
10:	E=BPT	(drive B pointer onto upper E bus)
11:	MARU+	(increment upper MAR)

bit 42   Upper Memory Chip Select Bar

0:	NOP13	(active low) (DEFAULT VALUE: chip selected)
1:	UCS_H	

bit 43   Upper Memory Write Enable Bar

0:	WEBU	(active low)
1:	NOP14	(DEFAULT VALUE: write disabled, =read mem)

bit 44   Upper Memory Chip Output Enable Bar

0:	NOP15	(active low) (DEFAULT VALUE: output enabled)
1:	UOE_H	

bits 45-47   Function ROM Select

000:	NOP16	(ROM does not drive C bus) (DEFAULT)
001:	SQR	(Square Root)
010:	RCP	(Reciprocal)
011:	user defined	
100:	user defined (LPROM)	
101:	user defined	"
110:	user defined	"
111:	user defined	"

bit 48 C Bus Tie

0:	NOP17
1:	CTIE

bits 49-52   Literal Inserter Control

0000:	NOP18	(SHIFTER DOES NOT DRIVE C BUS) (DEFAULT)
0001:	SAV1	(flags set1 to upper MSBs, LSBs no change)
0010:	SAV2	(flags set2 to upper MSBs, LSBs zeroed)
0011:	SAV3	(flags set3 to upper LSBs, MSBs no change)

		Mnemonic system example: I L Z L
0100:	ILZL	
0101:	IMNL	1st character: Insert literal
0110:	IMZL	
0111:	ILNL	2nd: Lsb's or Msb's get literal
1000:	ILZU	
1001:	IMNU	3rd: Zero other half or No change
1010:	IMZU	
1011:	ILNU	4th: Upper/Lower/Both busses get literal

1100:	ILZB	
1101:	IMNB	So the mnemonic ILZL means Insert the
1110:	IMZB	literal on the Lsb's and Zero out the msb's
1111:	ILNB	of the Lower A bus only.

bits 53-58 Conditional Multiplexer Select

000000:	FAL	(unconditionally false) (DEFAULT)
000001:	IO1	(I/O interrupt level 1)
000010:	IO2	(I/O interrupt level 2)
000011:	MZ	(multiplier zero)
000100:	MOVF	(mult overflow/int result more than 32 bits)
000101:	MUN	(multiplier underflow)
000110:	MNAN	(multiplier NaN - Not a Number)
000111:	MDEN	(multiplier denormalization trap)
001000:	AZ	(adder zero)
001001:	AOVF	(adder overflow)
001010:	ANAN	(adder NaN - Not a Number)
001011:	ADIF	(Inputs to adder differ by more than 2**63)
001100:	TRPS	(MOVF + MNAN + AOVF + ANAN + UALUO)
001101:	UIN1Z	(Upper INC1 = 0)
001110:	UIN2Z	(Upper INC2 = 0)
001111:	UIN3Z	(Upper INC3 = 0)
010000:	USO	(Upper shifter's shifted out bit =0)
010001:	LIN1Z	(Lower INC1 = 0)
010010:	LIN2Z	(Lower INC2 = 0)
010011:	LIN3Z	(Lower INC3 = 0)
010100:	UALUZ	(Upper ALU zero)
010101:	UALUN	(Upper ALU negative)
010110:	UALUO	(Upper ALU overflow)
010111:	UALUC	(Upper ALU carry)
011000:	unused	
011001:	unused	
011010:	unused	
011011:	UEVN	(Integer on upper C bus is even)(LSB = 0)
011100:	UR1_0	(UR1 bit[0]=1)(LSB)
011101:	UR1_1	(UR1 bit[1]=1)
011110:	UR1_2	(UR1 bit[2]=1)
011111:	UR1_3	(UR1 bit[3]=1)
100000:	TRU	(Unconditionally true)
100001:	NIO1	(not I/O interrupt level 1)
100010:	NIO2	(not I/O interrupt level 2)
100011:	NMZ	(not multiplier zero)
100100:	NMOVF	(not multiplier overflow)
100101:	NMUN	(not multiplier underflow)
100110:	NMNAN	(not multiplier NaN)
100111:	NMDEN	(not multiplier denormalization trap)
101000:	NAZ	(not adder zero)
101001:	NAOVF	(not adder overflow)
101010:	NANAN	(not adder NaN)
101011:	NADIF	(Inputs to adder do not differ by more than 2**63)

101100:	NTRPS	(MOVF + MNAN + AOVF + ANAN + UALUO = 0)
101101:	UIN1N	(Upper INC1 not = 0)
101110:	UIN2N	(Upper INC2 not = 0)
101111:	UIN3N	(Upper INC3 not = 0)
110000:	LSO	(Lower shifter's shifted out bit =0)
110001:	LIN1N	(Lower INC1 not = 0)
110010:	LIN2N	(Lower INC2 not = 0)
110011:	LIN3N	(Lower INC3 not = 0)
110100:	LALUZ	(Lower ALU zero)
110101:	LALUN	(Lower ALU negative)
110110:	LALUO	(Lower ALU overflow)
110111:	LALUC	(Lower ALU carry)
111000:	unused	
111001:	unused	
111010:	unused	
111011:	LEVN	(Integer on lower C bus is even)(LSB = 0)
111100:	UR1_28	(UR1 bit[28]=1)
111101:	UR1_29	(UR1 bit[29]=1)
111110:	UR1_30	(UR1 bit[30]=1)
111111:	UR1_31	(UR1 bit[31]=1)(MSB)(also used as immed addr flag)

bits 59-60   Branch Control (if condition not true, next adr = CAR+1)

00:	BR	(conditional branch) (DEFAULT)
01:	RET	(conditional return)
10:	CALL	(conditional call)
11:	MAP	(unconditionally use the MAP next address)

bits 61-63   Macrocode Support Mux Selects

000:	NOP19	(DEFAULT)
001:	RSEL	(override register select/bus tie fields)
010:	BRSEL	(override condition mux select field)
011:	SRSEL	(override Ebus/ETie ctrls with address source sel)
100:	ALSEL	(override ALU/Shift, reg sel, and bus tie fields)
101:	SHSEL	(override Barrel shift ctrl, reg sels, and bus ties)
110:	ISEL	(override increment and pointer ctrl fields)
111:	unused	

Lower ROM Fields - 64 bits

bits 0-4   Lower A Bus Select

00000:	NOP20	(no drive) (DEFAULT)
00001:	AL=R1	(lower A bus = lower R1)
00010:	AL=R2	
	|	
11001:	AL=R25	

```
11010:     AL=IN1  (lower A bus = lower INC1)
11011:     AL=IN2
11100:     AL=IN3
11101:     AL=CPT  (lower A bus = C pointer)
11110:     AL=DPT
11111:     AL=MBR
```

bits 5-9   Lower B Bus Select

```
00000:     NOP21
00001:     BL=R1
00010:     BL=R2
              |
11001:     BL=R25
11010:     BL=IN1
11011:     BL=IN2
11100:     BL=IN3
11101:     BL=FP*  (lower B = result of FP MULTIPLIER, LSB's)
11110:     BL=FP+  (lower B = result of FP ADDER, LSB's)
11111:     BL=MBR
```

bits 10-14   Lower C Bus Select

```
00000:     NOP22  (no load)  (DEFAULT)
00001:     R1=CL  (lower R1 loads from the lower C bus)
00010:     R2=CL
              |
11001:     R25=CL
11010:     IN1=CL (lower INC1 loads from the lower C bus)
11011:     IN2=CL
11100:     IN3=CL
11101:     CPT=CL (C pointer loads from the lower C bus)
11110:     DPT=CL
11111:     MBR=CL (Lower MBR loads from lower C bus)
```

bits 15-18   Lower ALU Select

```
0000:     NOP23
0000:     MOVLN   (SHIFTER_INPUT=A, flags unaffected)(DEFAULT)
0001:   ORL     (SHIFTER_INPUT = A or B)
0010:     ANDL    (SHIFTER_INPUT = A and B)
0011:     XORL    (SHIFTER_INPUT = A xor B)
0100:     MOVL    (SHIFTER_INPUT = A, flags affected)
0101:     NANDL   (SHIFTER_INPUT = A nand B)
0110:   NORL    (SHIFTER_INPUT = A nor B)
0111:     NOTL    (SHIFTER_INPUT = A')
1000:     INCL    (SHIFTER_INPUT = A + 1)
1001:     SETL    (set carry flag flip-flop)
1010:   ADCL    (SHIFTER_INPUT = A + B + cy flip-flop)
1011:     ADDL    (SHIFTER_INPUT = A + B)
1100:   NEGAL   (SHIFTER_INPUT = -A)
```

```
1101:     SUBL    (SHIFTER_INPUT = A - B)
1110:     SWBL    (SHIFTER_INPUT = A - B - cy flip-flop)
1111:     DECL    (SHIFTER_INPUT = A -1)

bits 19-22     Lower Shift Control (INPUTS COME DIRECTLY FROM ALU)

0000:     NOP24  (SHIFTER DOES NOT DRIVE C BUS)  (DEFAULT)
0001:     GNDCL  (C = 0  short C bus to GND, shift flags unaffected)
0010:     PASSL  (C = SHIFTER INPUT, shift flags unaffected)
0011:     SLOTL  (SHL with SH_OUT into LSB)
0100:     SLMSL  (circulate left with MSB into LSB)
0101:     SLCYL  (SHL with CY out of ALU into LSB)
0110:     SL0L   (SHL with 0 into LSB)
0111:     SL1L   (SHL with 1 into LSB)
1000:     SRLSL  (circulate right with ALU LSB into MSB)
1001:     SRCFL  (SHR with carry flip-flop into MSB)
1010:     SRSL   (SHR with SIGN FF into MSB)
1011:     SROTL  (SHR with SH_OUT FF into MSB)
1100:     SRSEL  (SHR with sign extension = MSB into MSB)
1101:     SRCYL  (SHR with CY out of ALU into MSB)
1110:     SR0L   (SHR with 0 into MSB)
1111:     SR1L   (SHR with 1 into MSB)

bits 23-24     Barrel Shifter Set-up

00:  SHROM  (shift using ROM control input) (DEFAULT)
01:  SHREG  (shift using control register input)
10:  L_ROM  (Load Bar ctrl reg, use ROM input to shift)
11:  L_REG  (Load Bar ctrl reg, use old reg value to shift)
NOTE: If you just want to load the reg, use L_ROM and NOP25 below.

bits 25-29     Barrel Shifter Control

00000:    NOP25 (Do not drive C bus)     (DEFAULT)
00001:   LCS1 (1 bit left circular shift)
00010:   LCS2 (2 bit left circular shift)
|
11111:    LCS31 (31 bit left circular shift)

bits 30-32 Lower Incrementable Registers Control

000:     NOP26  (no increment)(DEFAULT)
001:     LIN1+  (increment lower INC1)
010:     LIN2+  (increment lower INC2)
100:     LIN3+  (increment lower INC3)

bits 33-35 C,D Pointer Controls

000:  NOP27  (no change)  (DEFAULT)
001:  CIN_L  (load C increment size)
010:  DIN_L  (load D increment size)
```

```
011:  CDIN_L (load C,D increment sizes)
100:  CPT+   (increment C pointer)
101:  C+_CL  (increment C pointer, load C increment)
110:  D+_DL  (increment D pointer, load D increment)
111:  DPT+   (increment D pointer)

bits 36-37 Lower Memory Buffer Registers Control

00:  NOP28   (no action)  (DEFAULT)
01:  R1=DL   (Load lower R1 from lower D bus)
10:  R2=DL   (Load lower R2 from lower D bus)
11:  MB=DL   (Load lower MBR from lower D bus)

bits 38-39 Lower Memory Address Register Control

00:  MARL   (lower MAR drives lower address bus)  (DEFAULT)
01:  MAR=CL (drive lower addr bus and load MAR from lower C bus)
10:  MAR=EL (drive lower addr bus and load MAR from lower E bus)
11:  NMARL  (lower MAR does not drive lower address bus,
 and address pads go to high impedence)

bits 40-41 Lower E Bus Select

00:  NOP29  (no drive)  (DEFAULT)
01:  E=CPT  (drive C pointer onto lower E bus)
10:  E=DPT  (drive D pointer onto lower E bus)
11:  MARL+  (increment lower MAR)

bit 42 A Bus Tie

0:  NOP30
1:  ATIE  (tie the upper and lower A busses together)

bit 43 B Bus Tie

0:  NOP31
1:  BTIE  (tie the upper and lower B busses together)

bit 44    Lower Memory Chip Select Bar

0:  NOP33  (active low) (DEFAULT VALUE: chip selected)
1:  LCS_H

bit 45    Lower Memory Write Enable Bar

0:  WEBL      (active low)
1:  NOP34     (DEFAULT VALUE: write disabled, =read mem)

bit 46    Lower Memory Output Enable Bar

0:  NOP35  (active low) (DEFAULT VALUE: output enabled)
```

```
1:  LOE_H

bit 47        DONE FLAG
0:     NOP 36   (no action) (DEFAULT)
1:     DONE     (raise done flag)

bits 48-63     Next Address/Literal Field

       (Literals are composed of the bit pattern preceded by #)
```

Appendix B. *Newton-Raphson Inversion Routine*

```
; Newton - Raphson Inversion ROUTINE -- J CONTOIS
;
; LOCATION OF PASSED PARAMETERS:
;  UR25/LR25  : the number to be inverted
;
; RETURNS: result of inversion in UR23/LR23
;
; CHANGES: UR23 LR23 UR24 LR24 IN1L
;          FP* FP+ registers
;
; NOTES:  IN1L is the loop counter; it is
;         set to -3 at the start, ends at 0
;
; ANALYSIS: uses 31 clock cycles
;
; RESTRICTIONS:
;
; HISTORY:  Updated to rev. 3.0 microcode.
;
; ------------------------------------------------------------
NRI:  BU=R25  R23=CU  NANDU  PASSU   IMZL
              R23=CL  GNDCL        #1111111111110000;
;
; This line of code does the following:
;   UR23<=(UR25 NAND MASK),  LR23<=0
; The number to be inverted is used to generate the seed for
; this routine. A mask is inserted onto the lower A bus, passed
; to the upper A bus, and used to isolate the sign bit and the
; exponent bits of the number to be inverted (A). This exponent
; e has a binary value of e-1023 in IEEE format. This line of
; code inverts that e as well as masking out the mantissa.
; LR23 is cleared: it will be the LSB's of the seed's mantissa,
; which are 0. NOTE that the MSB's of the seed mantissa have
; been set to 1 by NANDing them with 0. This will be fixed
; later by clearing these with the literal ins.
;
; ------------------------------------------------------------
      BU=R23  R23=CU  ADDU   PASSU  IMZL
                             #0111111111100000;
;
; This line of code does the following:
;   UR23<=(UR23 + 1022)
; The exponent must be negated and decremented by 1 to get the
; exponent for the seed (-e-1). This is done by inverting the
; exponent and adding 1022. The inversion was done by NAND in
; the previous line, the addition of 1022 is done by this line
; of code. The result is -e-1 in IEEE format. The sign bit
; gets re-inverted back to its original value as a result of
; the carry out of the exponent addition.
; ------------------------------------------------------------
```

```
    BU=R25  R24=CU  RCP
            R24=CL  GNDCL;
;
; This line of code does the following:
;    UR24<=seed from function ROM, LR24<=0
; The function ROM is used to generate the 4 MSB's of the
; mantissa of the seed (initial guess at the result). It uses
; the lower 4 bits of the upper byte of the upper B bus (which
; correspond to the 4 MSB's in the mantissa of a floating
; point number) to choose the seed. LR24 is cleared, it will
; become the LSB's of a floating point "2".
; -------------------------------------------------------------

    AU=R23  BU=R24  R23=CU  ORU  PASSU  IMZL
    #1111111111110000;
;
; This line of code does the following:
;    UR23 <= [ UR24 OR (UR23 AND MASK) ]
; The seed's mantissa (UR24) is OR'd with the altered
; exponent -e-1 (UR23) to form the total seed, in proper
; IEEE floating point format. Note how the literal inserter
; has been used to clear the MSB's of the mantissa:
; the sign/exponent/4 MSB's are driven onto the upper A bus
; from UR23, then the literal inserter ALSO drives the upper
; A_bus. The literal chosen is 1111111111110000/00...00
; (all 0's in the LSB's by using IMZL).
; What happens is an AND: the literal bits that
; are 1 will not affect the bits driven from UR23, but the
; bits that are 0 will override the bits from UR23 due the
; operation of a precharged bus.
; -------------------------------------------------------------

            R24=CU          PASSU  IMZL
                            #0100000000000000;
;
; This line of code does the following:
;    UR24<=literal, = floating point 2
; A floating point "2" is needed by the algorithm. This
; line loads the upper R24 with the proper sign, exponent
; and MSB's. The Lower R24 holds the rest of the mantissa,
; which is all 0's. A IEEE floating point "2" looks like this,
; in UR24/LR24:
;
;         0                        positive sign
;          10000000000             "+1" IEEE
;                     0000/00 - 00 mantissa = 52 0's.
; -------------------------------------------------------------

    AU=R25  BU=R23  FP*L  TRU
    AL=R25  BL=R23        RLP2;
```

```
          FP*                           ILNL
                     IN1=CL  MOVL PASSL  #1111111111111101;
     ;
     ;  These lines of code do the following:
     ;     load the multplier with R25,R23, IN1L<=-3, jump into loop
     ;  on 3rd line These lines load the multiplier with A and X,
     ;  and do the multiply. The loop is entered on the third line
     ;  because the first 2 lines load the "X" calculated in the
     ;  previous iteration. For the first iteration "X" is the seed
     ;  created above. The line with FP* gets done before the jump
     ;  occurs due to pipeline delay. The lower incrementable register
     ;  is the loop counter, 4 loops are needed.
     ;
     ;   <<<<<<< TOP OF THE LOOP >>>>>>>
     ;
     ;  This loop does the following three calculations four times:
     ;          AX    A is the number to be inverted,
     ;        2-AX    X is the latest guess at 1/A.
     ;      X(2-AX) = new X for the next iteration.
     ;  Upon completion of the 4 iterations, X is the result: X=1/A.
     ; -------------------------------------------------------------

     RLP:  AU=R25  BU=FP*  FP*LD
           AL=R25  BL=FP*;

           FP*            TRPS  CALL
                          TRAP;
     ;
     ;  These lines load the multiplier with A and the latest
     ;  version of X. The line with FP* also checks the trap
     ;  conditions generated during the previous
     ;  floating point operation.
     ; -------------------------------------------------------------

     RLP2: AU=R24  BU=FP*  FP*D FP-L
           AL=R24  BL=FP*;

           FP-            TRPS  CALL
          TRAP;
     ;
     ;  These lines load the subtractor with 2 and AX from the
     ;  previous mult. The line with FP- also checks the trap
     ;  conditions generated during the previous
     ;  floating point operation.
     ; -------------------------------------------------------------

           AU=R23  BU=FP+  FP*L  LIN1N
           AL=R23  BL=FP+          RLP;

           FP*            TRPS  CALL
```

```
              LIM1+  TRAP;
;
; These lines load the multiplier with X and the result of
; the previous subtr. The first line also checks to see if
; the loop counter has not reached 0, in which case the
; program will jump to the top of the loop. The second line
; is done before the jump, due to the pipeline delay. The
; line with FP* also checks the trap conditions generated
; during the previous floating point operation, and
; increments the loop counter.
;
; >>>>>>> BOTTOM OF THE LOOP <<<<<<<
;

              R23=CU  FP*D  TRU RET
              R23=CL;
;
; This line of code does the following:
; R23<=F.P. Mult output, return to calling routine
; The final version of X is in the multiplier. This line
; puts that result into R23 and unconditionally returns to
; the calling routine. The last line of code in this routine
; (below) will be done before the return jump, so it is used
; to check for a trap condition occurring on the last multiply.
; -------------------------------------------------------------

                     TRPS  CALL
                     TRAP;

TRAP: nop;     placeholder for unwritten TRAP routine
end;
; -----------------------------------------------------------
```

B.1 Dot Product Routine

```
; DOT PRODUCT MICROCODE SUBROUTINE -- J CONTOIS
;
; LOCATION OF PASSED PARAMETERS:
;    MAR and CPTR: Pointer to C vector
;    BINC        : distance between elements of C vector in memory
;    APTR        : Pointer to A vector
;    AINC        : distance between elements of A vector in memory
;    LINC1       : -N    the length of the vectors
;    UR22        : pointer to memory location for result (optional)
;
; RETURNS: result in MBR, memory address of result in MAR
; CHANGES: LINC1, R1, MBR, MAR, APTR, CPTR, FP* and FP+ REGISTERS
; NOTES:  R1 holds C vector elements, MBR holds A vector elements
; ANALYSIS: 2N+6 clock cycles for vectors of length N
; RESTRICTION: N must be greater than or equal to 1
;
; HISTORY: Updated to 3.0 revision of the FPASP microcode.
;          Additional comments were added by G. Morris, one
;          of the EE588 students.
;
;-----------------------------------------------------------------
;
; At this point, MAR points to the first C vector element, APTR and CPTR
; point to the A and C vectors, the A and C incr registers have the
; distance between successive elements, and LINC1 has the vector size.
;
; This line of code accomplishes the following actions:
; clear R25, load R1 <- C[0], load MAR <- A[0] addr, inc A and C ptrs

DOTP: R25=CU  GNDCU   APT+        R1=DU  MAR=EU  ETIE E=APT
      R25=CL  GNDCL   CPT+        R1=DL  MAR=EL  ;

; Now we have C[0] in R1, MAR points to A[0], R25 contains zero, the
; A and C pointers point to the second vector elements.
;
; This line of code accomplishes the following actions:
; clear FP adder regs, load MBR <- A[0], load MAR <- C[1] addr, inc C ptr

      AU=R25 BU=R25  FP+L            MB=DU MAR=EU   ETIE
      AL=R25 BL=R25        CPT+      MB=DL MAR=EL   E=CPT;

; MBR contains A[0], R1 contains C[0], the FP+ regs contain
; zero.  MAR has the address of C[1] and C ptr points to B[2], A ptr to A[1]
;
; This line of code accomplishes the following actions:
; FP add 0's, load FP multiplier regs with A[0] and C[0],load R1 <- C[1]
; load MAR <- A[1] addr, inc A ptr, and increment the loop counter

      AU=MBR  BU=R1  FP*L FP+ APT+   R1=DU  MAR=EU  ETIE E=APT
      AL=MBR  BL=R1           LIN1+  R1=DL  MAR=EL;
```

```
; At the top of this loop, the conditions are as follows:
; the multiplier registers contain the next two elements to be multiplied
; R25 has the last product, MBR has the next+1  A, and R1 has the next+1 C
; the FP+ registers contain the partial sum, the A and C ptrs point to
; the next+2 A and C elements.  NOTE:  The conditional branch instruction
; which intuitively belongs on the next line of code is physically on this
; line of code to accomodate the pipeline register delay.  Even if the branch
; fails, e.g., we fall thru, the next instruction WILL be executed.
;
; This line of code accomplishes the following:
; multiply, put R25 (the previous product) and the sum so far (the result of
; the current FP adder registers) back into the adder regs, load MBR <- next A
; load MAR <- next C address, and inc C ptr

DPLP: AU=R25 BU=FP+  FP* FP+L        MB=DU MAR=EU ETIE       LIN1N BR
      AL=R25 BL=FP+       CPT+       MB=DL MAR=EL      E=CPT DPLP;

; At this point, the multiply has finished, the next C addr is in the
; MAR, R1 and MBR have the new A and C values, and the adder is loaded
; with the partial sum and the last product.
;
; This line of code accomplishes the following:
; Load R25 <- product, load FP* regs from R1 and MBR, add the
; FP+ partial sum, load R1 <- next C value, load MAR <- next A addr
; inc A ptr, inc the loop counter, if any errors, handle them
; NOTE: There is a branch after this instruction via the previous instr!

      AU=MBR BU=R1 R25=CU FP*LD FP+ APT+ R1=DU  MAR=EU ETIE E=APT TRPS CALL
      AL=MBR BL=R1 R25=CL LIN1+          R1=DL  MAR=EL         TRAP;

; At this point, the last product is already in R25, and the partial
; sum has been computed and is available at the adder output.
;
; This line of code accomplishes the following:
; Load the last product term and the partial sum into the FP adder regs

      AU=R25 BU=FP+ FP+L
      AL=R25 BL=FP+;

; The adder is loaded with the required operands.
;
; This line of code does the addition, and loads the MAR with the
; destination address contained in R22.  It also does an unconditional
; return, but as before, there is a one cycle delay due to the pipeline

      AU=R22 PASSU  FP+                       MAR=CU   TRU RET
             PASSL                            MAR=CL  ATIE;

; At this point, the FP+ has finished adding, and the final result is
; waiting at the output.
```

```
;
; This instruction loads MBR with the final dot product
; and then returns to the calling microcode routine (note, the
; ret instruction was actually stated in the previous line of code).

            MBR=CU FP+D
    MBR=CL        ;
;
;  The routine returned to must do the write to memory of the result of DOTP
;
TRAP: nop   ;placeholder for trap routine
end;
; ---------------------------------------------------------------------------
```

Bibliography

Cal86. Scott, Walter S., and others, "1986 VLSI Tools: Still More Works by the Original Artists," *User's Manual,* Computer Science Division, EECS Department, University of California at Berkeley, 1986.

Duk88. Dukes. Michael A., "A Multiple-Valued Logic System for Circuit Extraction to VHDL 1076-1987," *MS Thesis, AFIT/GE/ENG/88S-1,* School of Engineering, Air Force Institute of Technology (AU), September 1988.

Fle88. Fleckenstein, Donald C., chairman, "IEEE Standard VHDL Language Reference Manual," Institute of Electrical and Electronics Engineers, Inc., New York, New York, 1988.

Fre88. Fretheim, Erik J., *Internal Communication on the double precision multiplier* Air Force Institute of Technology (AU), November 1988.

Fuj85. Fujiwara. Hideo, *Logic testing and Design for Testability* The MIT Press, Cambridge, Massachusetts, 1985.

Gal87. Gallagher, David M., "Rapid Prototyping of Application Specific Processors," *MS Thesis, AFIT/GE/ENG/87D-19,* School of Engineering, Air Force Institute of Technology (AU), December 1987.

Gla85. Glasser, L. A. and Dobberpuhl, D. W., *The Design and Analysis of VLSI Circuits.* Addison Wesley Press, Reading, Massachusetts, 1985.

Gun88. Gunn, Lisa, "At 100MFLOPS, the Fastest DSP Chip Ever," *Magazine article,* Electronic Design, A VNU Business Publication, Southeastern, Pennsylvania, 13 October 1988.

Hau87. Hauser. Robert S., "Design and Implementation of a VLSI Prime Factor Algorithm Processor," *MS Thesis, AFIT/GE/ENG/87D-5,* School of Engineering, Air Force Institute of Technology (AU), December 1987.

Hen84. Hennessey, J. L., "VLSI Processor Architecture," *IEEE Transactions on Computers,* C-33 pp. 1221-1246, December 1984.

Hit88. , no author listed. "Will Flash Memories Replace Today's Memory Chips?," *Magazine article,* High Technology Business, Infotechnology Publishing Corporation, New York, New York, August 1988.

Lin85. Linderman. Richard W., and others, "CSTAT SIM File Checker for CMOS Chips," *Student handout,* Air Force Institute of Technology Department of Electrical and Computer Engineering, Wright- Patterson AFB, OH, 1988.

Lin88. Linderman. Richard W., editor, "Compendium of Project Reports Submitted for EENG 588: Computer Systems Architecture Summer Quarter 1988," *AFIT Technical Report* Air Force Institute of Technology (AU), October 1988.

Lin88-2. Linderman. Richard W. and others, "Design and Application of an Optimizing XROM Silicon Compiler," *Paper submitted to the IEEE Journal on Computer Aided Design for Integrated Circuits,* October 1988.

Man82. Mano, M. Morris, *Computer System Architecture, Second Edition,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

May88. Mayhew, Brett F., "Double Precision Floating Point Adder and Subtractor," *Class Project Report,* Air Force Institute of Technology (AU), September, 1988.

Mit86. Mitchell. H.J., editor, *32-Bit Microprocessors,* McGraw-Hill Book Company, New York, New York, 1986

MOS88. no author listed, "MOSIS User Manual Release 3.0," University of Southern California, 1988.

Mot88. *Technical Summary, 32-Bit Third-Generation RISC Microprocessor,* Motorola Inc, Literature Distribution, Phoenix, Arizona, 1988.

Per88. *Product Brochure, P4C1257 Ultra High Speed 256Kx1 CMOS Static RAM,* Performance Semiconductors, September 1987.

Qua86. Quarles. T., and others, "Spice 3B1 User's Guide," *Air Force Institute of Technology Department of Electrical and Computer Engineering, Student Handout,* Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 1988

Ros85. Rossbach. Paul C., "Control Circuitry for High Speed VLSI Winograd Fourier Transform Processors," *MS Thesis, AFIT/GE/ENG/85D-35,* School of Engineering, Air Force Institute of Technology (AU), December 1985.

She86. Shephard. Carl G., "Integration and Design for Testability of a High Speed Winograd Fourier Transform Processor," *MS Thesis, AFIT/GE/ENG/86D-46,* School of Engineering, Air Force Institute of Technology (AU), December 1986.

Jav87. Javed, Mohammad S., "Board Level Development of the AFIT VME Design Board," *MS Thesis, AFIT/GE/ENG/87J-3,* School of Engineering, Air Force Institute of Technology (AU), December 1986.

Spa86. Spanburg, Craig S., "Laser Programming Integrated Circuits," *Master's Degree Thesis, AFIT/GE/ENG/87D-62,* School of Engineering, Air Force Institute of Technology (AU), December 1987.

Ter86. Terman, Chris, "Esim: An Event Driven Switch Level Simulator," *Berkeley CAD Tools User's Manual,* University of California at Berkeley, 1986.

Til88. Tillie, John J., "Laser Programmable Read Only Memories," *Master's Degree Thesis, AFIT/GE/ENG/88D-56,* School of Engineering, Air Force Institute of Technology (AU), December 1988.

Wes85. Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design,* Addison Wesley Press, Reading Massachusetts, 1985.

Vita

Captain John H. Comtois [redacted]. He attended [redacted] He then attended Carnegie- Mellon University and enrolled in the Air Force via the Air Force Reserve Officers Training Course. Upon graduating in May 1983 he received a B.S.E.E. degree and a commission in the Air Force as a Second Lieutenant. His first assignment was as programmer/analyst and later as maintenance branch chief at the Command and Control Systems Office at Tinker AFB, Oklahoma. While there he was promoted to First Lieutenant and received a regular commission in the Air Force. After his graduation from the Air Force Institute of Technology he will begin an assignment with the 84 Radar Evaluation Squadron at Hill AFB, Utah.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/88D-5	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering; 6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Proj. Agency; 8b. OFFICE SYMBOL *(If applicable)* DARPA/ISTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20332	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Architecture and Design of a Laser Programmable Double Precision Floating Point Application Specific Processor (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
John H. Comtois, B.S., Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Thesis	FROM ______ TO ______	1988 December	151

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	06	
09	01	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer Architecture, Laser Programming, VLSI Microprocessor, Integrated Circuits, VHSIC, VMEbus

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Chairman: Richard W. Linderman, PhD., Captain, USAF
Assistant Professor of Electrical Engineering

[illegible]
10 Jan 89

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT.] ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)*	22c. OFFICE SYMBOL
Joseph DeGroat, Major, USAF	513-255-3576	AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. ABSTRACT

Numerous signal processing systems in the Department of Defense and industry would benefit from a microprocessor tailored to their specific applications. This thesis effort describes the architecture and design for a 64 bit application specific processor (ASP) which combines the power of double precision floating point hardware with the flexibility of a laser programmable microcode store. This floating point ASP (FPASP) contains a variety of circuits which efficiently perform digital signal processing and other applications requiring double precision floating point arithmetic.

This thesis describes a new rapid rapid prototyping methodology for ASPs. The user provides an algorithm which is translated into microcode, tested on a software model, and then cut into the laser PROM of a blank FPASP chip. With this methodology the prototyping of an ASP can be completed in a matter of days, and no hardware design is involved. The programmed FPASP can then be mounted on a circuit board and placed in a host processor to act as a hardware accelerator for computationally intense programs. The FPASP also supports a macro assembly language which can be partially user-defined. So the FPASP can be tailored to higher level applications such as operating system support.

The FPASP has been designed to support common software structures. It contains registers for loop indexing, and addressing into matrices. The FPASP also contains a subroutine stack 16 words deep, which can be extended into the external memory for an additional 1023 words to support recursive microcode routines. The FPASP will contain 180,000 CMOS transistors on a chip 0.35 inches on a side. It is designed to operate at 25 MHz, and at that speed it will be capable of performing 25 million floating point operations a second.

The FPASP architecture consists of two 32 bit processors which can operate independently for integer operations, or in tandem for double precision floating point operations. Overlapping register sets on each datapath and ties between the two datapaths provide a high degree of interconnectivity, allowing efficient internal data transfer between the 66 32 bit registers and the processing circuits.